

AD-A102 180

MITRE CORP MCLEAN VA

F/G 5/1

CANDIDATE R&D THRUSTS FOR THE SOFTWARE TECHNOLOGY INITIATIVE.(U)

MAY 81 S T REDWINE, E D SIEGEL, G R BERGLASS F19628-81-C-0001

NTR-81W00160

NL

UNCLASSIFIED

1-1-81  
40  
4-12-80



**LEVEL II**

**12**

**AD A102180**

# **CANDIDATE R&D THRUSTS FOR THE SOFTWARE TECHNOLOGY INITIATIVE**



**DTIC FILE COPY**

**Department of Defense  
May 1981**

**DTIC  
SELECTE  
JUL 30 1981**

**DISTRIBUTION STATEMENT A**  
Approved for public release;  
Distribution Unlimited

**D**

**81 7 29 010**

**+**

# CANDIDATE R&D THRUSTS FOR THE SOFTWARE TECHNOLOGY INITIATIVE

Department of Defense



May 1981

S JUL 30 1981 D  
D

Prepared with the Assistance of the MITRE Corporation

Samuel T. Redwine, Jr.  
Eric D. Siegel  
Gilbert R. Berglass

Foreword By  
Joseph C. Batz  
Office of the Under Secretary of Defense  
for Research and Engineering  
(Electronics & Physical Sciences)

Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Avail and/or	
Dist	Special
A	

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. <b>AD-A102180</b>	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle)  CANDIDATE R&D THRUSTS FOR THE SOFTWARE TECHNOLOGY INITIATIVE		5. TYPE OF REPORT & PERIOD COVERED  Final
7. AUTHOR(s)  Samuel T. Redwine, Jr., Eric D. Siegel, and Gilbert R. Berglass		6. PERFORMING ORG. REPORT NUMBER <b>MTR-81W00160</b>
9. PERFORMING ORGANIZATION NAME AND ADDRESS  MITRE 1820 Dolley Madison Blvd. McLean, VA 22102		8. CONTRACT OR GRANT NUMBER(s)  F19628-81-C-0001
11. CONTROLLING OFFICE NAME AND ADDRESS  Office of Electronics and Physical Sciences Rm 3D1079 The Pentagon Washington, DC 20301		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS  8920
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		12. REPORT DATE <b>May 1981</b>
		13. NUMBER OF PAGES <b>xiv + 202</b>
		15. SECURITY CLASS. (of this report)  Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report)  Approved for Public Release; Distribution Unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES  Foreword by Joseph C. Batz, Office of the Under Secretary of Defense for Research and Development (Electronics and Physical Sciences)		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)  Software, Research and Development Planning, Technology Base, DoD Software Technology Initiative, Productivity, Reliability, Maintainability, Technology Transfer, Computers		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)  This document is the first iteration towards a technical plan for the DoD Software Technology Initiative, and is intended for review and comment. The background of the Initiative and DoD's historical difficulties with software are covered. Tentative candidates for R&D support are discussed in the sequence of their potential for significant incremental payoff -- short-term, (less than 4 years), medium-term (4-7 years), and long-term (more than 7 years). More detailed discussion of the candidates and a list of ideas tentatively rejected are included in appendices. Reviewers should comment using the questionnaire.		

## FOREWORD

DoD's software problems have been identified and reported in many previous studies. Expanded computer utilization, encouraged by extraordinary hardware advances will amplify these software problems by an order of magnitude over the next decade. New technologies, new methodologies, and new management approaches must be found to overcome them.

THE SOFTWARE TECHNOLOGY INITIATIVE is a joint DoD Tri-service effort to pursue the rapid transfer of proven technologies and the development of new technologies to obtain major improvements in software productivity, reliability and maintainability. This document has been prepared to lay the framework for the process of identifying, assessing, selecting and initiating research and development thrusts to solve problems common to all services and DoD components.

It is recognized that individual service efforts are underway to address some of these problems from a service-unique perspective. The objective in the Initiative is to seek out common elements and attack them jointly.

The proposed efforts and research activities in this document have been compiled from recommendations in published reports and from suggestions submitted to DoD in response to recent public announcements of the Software Technology Initiative. As described in this document, candidate thrusts may appear abstract, especially in areas where experience, knowledge, or capability already exists. However, it was intended that thrusts first be defined in a generic way in order that we not initially bind our approach to specific capabilities and implementations with which we may happen to be most familiar.

Potential efforts must be prioritized according to the problems they address, and defined in much greater detail before decisions can be made for additional funding and management support. In the process of prioritization, considerable thought must be given to the magnitude of the problems each effort addresses, as well as the impact the expected results will achieve. Selection will ultimately depend on the potential for reducing or eliminating recognized problems. Specific well-defined targets with scheduled milestones must be established to create a supportable program.

No claim is made that the potential thrusts described are all-encompassing; therefore, recommendations for additional thrusts are invited. Knowledgeable constructive criticism of the proposed thrusts is sought. The report itself provides a basis and mechanism for such comments and recommendations.

Responses will be evaluated by the Research and Development Technology Panel of the Management Steering Committee for Embedded Computer Resources. These results will be used in updating the Defense Computer Resources Technology Plan, and will serve as the basis for establishing DoD-sponsored research and development programs for the Software Technology Initiative.

Joseph C. Batz  
Office of the Under Secretary of  
Defense for Research and Engineering  
(Electronics & Physical Sciences)

## EXECUTIVE SUMMARY

The Software Technology Initiative (STI) is a new Department of Defense initiative aimed at order-of-magnitude improvements in DoD's capabilities to develop and maintain software. The key themes of the Initiative are productivity, reliability and maintainability, and technology transfer.

The need for the Initiative is evident from both the history of DoD problems with software and projections of tremendous increases in the importance and cost of software to DoD over the next decade. For example, the Electronic Industries Association (EIA) forecasts that 32 billion dollars will be spent by DoD on embedded software in FY 1990 alone.

The STI was first proposed by Dr. Ruth Davis during Congressional testimony in April, 1979. Responsibility for planning for the Initiative rests with the Office of Electronics and Physical Sciences under the OUSDRE (R&AT) and the Research and Development Technical Panel under the Management Steering Committee for Embedded Computer Resources.

R&D to be performed will be managed through the same DoD organizations that currently administer such efforts but coordinated under OUSDRE with the services, agencies, and laboratories, the Ada Joint Program Office, DARPA, the Very High Speed Integrated Circuit Program (VHSIC), the Advisory Group on Electronic Devices (AGED), and the DoD Computer Security Initiative. Plans are being formulated to establish a formal mechanism to obtain inputs from industry and academia in the subject areas of information sciences and software technology.

Increasingly lower hardware costs, the maturation of software engineering, and the introduction of Ada and the Ada Programming

Support Environment make this a propitious time to launch an Initiative to reduce the cost, time, and risk associated with software development and maintenance, while increasing the utility of operational systems. Both new and existing systems are targets for improvement.

Approximately eighty potential R&D thrust areas have been identified. Of these, forty are suggested R&D candidates for the STI. They are organized according to the time (under 4 years, 4-7 years, over 7 years) when their principal benefits will begin. A thrust is an effort to provide or apply a technical capability involving hardware, software, documentation, procedure, management, or education.

Short-term candidates emphasize technology transfer, standardization of software environments, tools, packages, workstations, and preliminary results from thrusts whose main payoffs occur later. Candidates with short-term payoffs include identifying and transferring techniques and competencies of superior software personnel, establishing the framework for an integrated software support environment, producing reusable Ada packages for common usage areas (e.g. graphics), and rapid prototyping.

Candidates with medium-term payoffs emphasize requirements and design methods, integrated software tool sets, and personnel and management improvements. Examples include data flow techniques, intensive advanced training for software personnel, and acquisition management support systems.

Long-term candidates are directed at problems without currently available solutions. Multi-computer system design, intelligent systems, and synthesis of techniques are emphasized.

Although the set of candidates may change, as planning and coordination proceed, a sound technical basis exists for the Initiative. Special emphasis will be needed to ensure successful technology

transfer and human engineering, because widespread use of the results within DoD is essential if the payoffs are actually to be received. In addition, provisions must be made for continuing maintenance and improvement of Initiative products. In brief, the technical basis exists; with adequate R&D effort and proper attention to technology transfer, the Software Technology Initiative will achieve the needed major improvements.

## TABLE OF CONTENTS

	<u>Page</u>
FOREWORD	iii
EXECUTIVE SUMMARY	v
LIST OF ILLUSTRATIONS	xiv
 1.0 INTRODUCTION	 1
1.1 Background	1
1.1.1 The Software Problem	1
1.1.2 Timeliness of the STI	5
1.1.3 History	5
1.1.4 Relationship to Other Efforts	9
1.2 Definitions	10
1.3 Themes	11
1.4 Candidate Lifecycle	12
1.5 Document Organization	14
 2.0 THRUSTS WITH SHORT-TERM PAYOFFS (UNDER 4 YEARS)	 15
2.1 Introduction	15
2.2 Candidates	15
2.2.1 Technical	17
2.2.2 Managerial	19
2.2.3 Personnel-Related	19
2.2.4 Continuity-Related	20
2.3 Chapter Summary	20
 3.0 CANDIDATES WITH MEDIUM-TERM PAYOFFS (4 - 7 YEARS)	 21
3.1 Introduction	21
3.2 Candidates	21
3.2.1 Technical	21
3.2.2 Managerial	24
3.2.3 Personnel-Related	24
3.2.4 Continuity-Related	25
3.3 Chapter Summary	25
 4.0 LONG-TERM CANDIDATES (MORE THAN 7 YEARS)	 27
4.1 Introduction	27
4.2 Candidates	27
4.2.1 Technical	27
4.2.2 Managerial	30
4.2.3 Personnel-Related	30
4.2.4 Continuity-Related	31
4.3 Chapter Summary	31

## TABLE OF CONTENTS (Continued)

	<u>Page</u>
5.0 SUMMARY	32
A. DESCRIPTION OF CANDIDATES	34
A.1 Technical	38
A.1.1 General	38
A.1.1.1 Integrated Software Support Environment	38
A.1.1.2 Ada Package Sets for Common Usage Areas	41
A.1.1.3 System Dictionary/Directory	43
A.1.1.4 Set(s) of Tools Covering Entire Lifecycle	46
A.1.1.5 Software Engineer's Support System	49
A.1.1.6 Programmer Workstation	52
A.1.1.7 Useful Measures of Software Quality	55
A.1.1.8 Multiple Representations of Software	58
A.1.1.9 Earliest Possible Error Detection	60
A.1.1.10 Configuration Independence	63
A.1.2 Conception/Feasibility	66
A.1.2.1 Rapid Simulation	66
A.1.3 Requirements	69
A.1.3.1 Rapid Prototyping	69
A.1.3.2 Application Domain Expertise	72
A.1.3.3 Data Validation	74
A.1.3.4 Built-In Testing	77
A.1.3.5 Forgiving Systems	80
A.1.3.6 User-Oriented Requirements Interfaces	83
A.1.3.7 Complex Knowledge-Based Systems	87
A.1.4 Design	91
A.1.4.1 Data Flow Approach	91
A.1.4.2 Self-Interfacing Software	93
A.1.4.3 Predicate Approach	95
A.1.4.4 Exception Handling	98
A.1.4.5 Distributed Functions and Resources	101
A.1.4.6 Suitable Communication Interconnection	104
A.1.5 Programming	108
A.1.5.1 Transform Software to Improve Quality	108
A.1.5.2 Formal Verification of Large Systems	111
A.1.6 Testing	114
A.1.6.1 High-Confidence Software Testing	114
A.1.7 Operations	117
A.1.7.1 Facilitating System Evolution	117
A.1.7.2 Impact Analysis of Proposed Change	120

## TABLE OF CONTENTS (Continued)

	<u>Page</u>
A.2 Managerial	123
A.2.1 Acquisition Manager's Support System	123
A.2.2 Software Technology-Compatible Acquisition	127
A.2.3 Technology Transfer in the Software Area	130
A.3 Personnel-Related	133
A.3.1 Superperformer Competencies	133
A.3.2 Intensive Advanced Programmer Training	136
A.3.3 Programmer Laboratory	139
A.3.4 Personnel Independence	142
A.3.5 Improved Education About Software	144
A.3.6 User Programming	146
A.4 Continuity-Related	149
A.4.1 Voice Replaces Text	149
A.4.2 Built-In Training and Documentation	152
 B. OTHER IDEAS	 155
B.1 Technical	155
B.1.1 General	155
B.1.1.1 Presentation and Manipulation	155
B.1.1.2 Rigorous Documentation	155
B.1.1.3 Conflict Recognition Among Representations	156
B.1.1.4 Exploratory Systems Applications of VHSIC	156
B.1.1.5 Military Information Utility	156
B.1.1.6 Multiple Classes of Service	157
B.1.1.7 Standard Real-Time Operating System	157
B.1.2 Requirements	157
B.1.2.1 Rapid Derivation of Requirements	157
B.1.2.2 Transform Informal to Formal Requirements	157
B.1.2.3 Requirements Languages Translation	158
B.1.2.4 Weakest Possible Requirements Description	158
B.1.3 Design	158
B.1.3.1 Derivation of Software from Specifications	158
B.1.3.2 Very High Level Languages	158
B.1.3.3 Component Tailoring and Interfacing	159
B.1.3.4 Publication of Standard Designs	159
B.1.3.5 Data Structure and Abstraction	159
B.1.4 Programming	159
B.1.4.1 Code Skeletons	159
B.1.4.2 Graph-Oriented Language	159
B.1.4.3 Generating Assertions from Requirements	160
B.1.4.4 Transform to Satisfy Physical Constraints	160
B.1.4.5 Man-Machine Quality Improvement Team	160

# TABLE OF CONTENTS (Continued)

	<u>Page</u>
B.1.4.6 Application Generators	160
B.1.4.7 Reusable Software	160
B.1.4.8 Actor Languages	160
B.1.5 Testing	161
B.1.5.1 Static Analysis of Software	161
B.1.5.2 Generating Test Data from Requirements	161
B.1.5.3 Generating Test Data to Violate Assertions	161
B.1.5.4 Testbed Facilities	161
B.1.6 Operations	162
B.1.6.1 Construction for Future Evolution	162
B.1.6.2 Modification of Large Systems	162
B.2 Managerial	162
B.2.1 General	162
B.2.1.1 Model Contracts for Buying Software	162
B.2.1.2 Maximizing DoD Rights to Software	162
B.2.1.3 Multiplying Expert Effectiveness	163
B.2.2 Conception/Feasibility	163
B.2.2.1 Quick Look Feasibility/Evaluation	163
B.3 Continuity-Related	163
B.3.1 General	163
B.3.1.1 Completely Captured Software	163
B.3.1.2 Multi-person Machine Mediated Programming	164
B.3.1.3 Totally Visible Software	164
B.3.1.4 Systems that Never Forget	164
C. SOFTWARE PROBLEM AREAS	165
C.1 Technical	166
C.1.1 Flawed and Conflicting Standards	166
C.1.2 Inappropriate Constraints	169
C.1.3 Poor Definition of Goals and Measures	169
C.1.4 Faulty Design	171
C.1.5 Incorrect Selection and Use of Languages & Packages	172
C.1.6 Poor Use of Implementation Tools	172
C.1.7 Inferior Testing Methodology	172
C.1.8 Unsatisfactory Product Evaluation and Follow-up	173
C.2 Managerial	173
C.2.1 Weak Project Leadership and Coordination	173
C.2.2 Poor Monitoring & Prediction of Schedules & Budgets	174
C.2.3 Unsatisfactory Project Control	175
C.2.4 Flawed Methodology for the Acquisition Process	175
C.3 Personnel-Related	176
C.3.1 Problems Finding and Keeping Qualified Personnel	177

## TABLE OF CONTENTS (Continued)

	<u>Page</u>
C.3.2 Unsuitable Competence Measures	178
C.3.3 Poor Exploitation of Personnel	179
C.4 Continuity-Related	179
C.4.1 Ambiguous, Unclear, Incomplete Communication	179
C.4.2 Slow, Outdated Communications	180
C.4.3 Lack of a Project History	180
C.4.4 Poor Phase-to-Phase Continuity	180
C.5 Bibliography	180
 D. SUMMARY OF SOME REVIEWED STUDIES	 183
 E. EVALUATION CONSIDERATIONS	 187
E.1 Benefits	187
E.2 Cost of R&D Thrusts	193
E.3 Types of Relationships Among Candidates	193
E.4 References	194
 F. SOFTWARE TECHNOLOGY INITIATIVE QUESTIONNAIRE	 195

## LIST OF ILLUSTRATIONS

<u>Figure Number</u>		<u>Page</u>
1	Software and Hardware Costs for Embedded Systems	2
2	Typical Thrust Lifecycle	13
3	Short-term Candidates	16
4	Medium-term Candidates	22
5	Long-term Candidates	28
6	Summary of Candidates' Tentative Status	35
7	Problem Areas	167
8	Theoretical Calculation of Benefits	189
9	Software Lifecycle	190

## 1.0 INTRODUCTION

The Software Technology Initiative (STI) is a tri-service, DoD-managed program to achieve an order-of-magnitude improvement in software productivity, reliability, and maintainability for military systems. Innovative management and engineering tools and techniques will be developed, and proven tools and techniques will be introduced into the software lifecycle throughout the Services and the participating Agencies. The goal is the development of a standard, coherent set of capabilities to support and enhance software development and maintenance. This document outlines the need for the STI and begins the process of choosing the initial R&D thrusts for the Initiative. To that end, DoD software problems, potential thrust areas, and any existing or proposed tools and techniques to manage the problems are described. Chapter 1 contains background material, definitions, and themes.

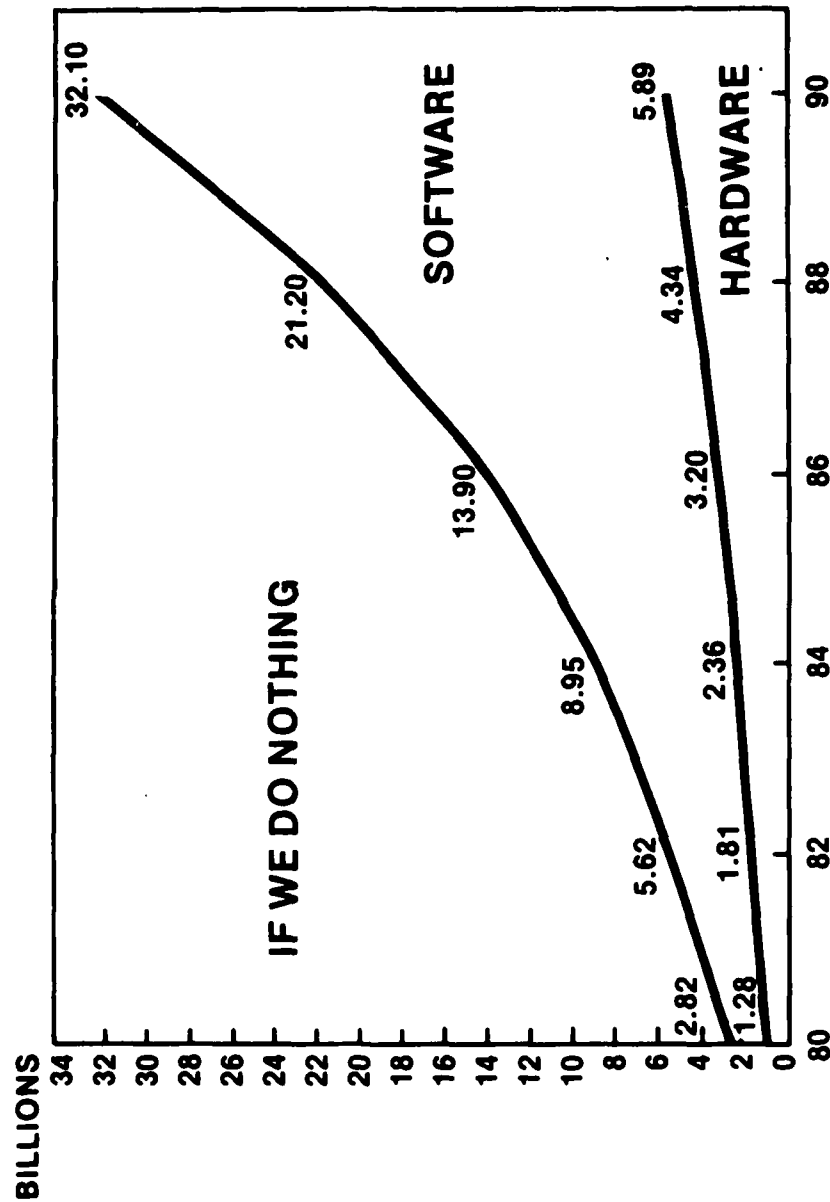
### 1.1 Background

This section presents background material on the need for and the history of the STI, along with information about its relationship to other ongoing Federal efforts.

#### 1.1.1 The Software Problem

The principal reason for a Software Technology Initiative is illustrated in Figure 1. The annual bill for software is rising dramatically as more computers find their way into systems and equipment. For example, DoD's bill for software in 1980 alone was estimated to exceed \$3 billion [DoD Annual Report FY81, Jan. 29, 1980, p. 245], and Figure 1 shows that it is forecast to exceed \$30 billion a year for embedded software in 1990. The indirect costs of software problems and schedule slippages are far greater than the figures show, because of the impact of faulty or delayed software on systems in which it is embedded, and because of the continuing high maintenance costs for software--especially for software with a

# DOD EMBEDDED COMPUTER SOFTWARE/HARDWARE



Source: Electronic Industries Association.

Figure 1: Software and Hardware Costs for Embedded Systems

problem-plagued development history. Many software systems have grown so complex that it has become virtually impossible to use existing methods to predict the costs and schedules of software development and maintenance projects.

Costs and schedules are not the only driving forces behind the STI. As software has become an increasingly large part of military information and weapons systems its reliability and availability have become crucial issues. Software systems must often operate in unpredictable and hostile environments, on battlefields, in combat aircraft, and in ships and submarines. Systems that must survive and operate in such environments are of significant importance; software failures may adversely affect the outcome of conflicts.

Unfortunately, software often doesn't even arrive at the stage where its reliability is tested, because failures in the requirements analysis, specification, and design process have left it in a state where it simply doesn't work or meet the need for which it was supposedly designed. Without a successful Software Initiative, The U.S. may find itself in a vulnerable position. As stated by the Software Methodology Panel in What Can Be Automated? (a computer science and engineering research study sponsored by NSF, edited by Bruce Arden [MIT Press, 1980]):

"... for the last 20 years the United States has enjoyed a commanding lead in computer power; the U.S.S.R. has had much less. However, Soviet programmers take their work seriously, with the result that they are relatively few but highly disciplined, while the United States has a great many undisciplined programmers.... No matter how good machines are, they cannot make up for sloppy programming. It is all too easy for poorly conceived programs to waste machine time even faster than hardware technology can provide it, and to end up with software systems which are beyond intellectual control and responsible management because of the lack of discipline in their development.... We could then face a software gap more serious than the missile gap of some years ago." [pp. 795, 796]

A critical shortage of competent software personnel aggravates the software problem. Although the number of programmers, about 240,000 in the U.S. in 1980, is growing, the need for programmers, as well as software engineers, designers, and analysts, is growing even faster. Competition in industry for software personnel drives up salaries, which will attract more people to computing careers; however, high salaries in industry entice personnel from lower-paying positions in universities and government, resulting in fewer professionals to train new software people and government's greater dependence on outside expertise.

The STI approaches the personnel shortage problem on several fronts. Principally, it aims to increase the productivity of existing personnel by giving them better training, hardware, software, procedures, and management. It seeks ways to identify or create superior performers, and to increase project productivity by finding better ways to use people, by standardizing procedures, and by facilitating the re-use of existing software. Ultimately, it seeks to change the composition of the software team to highly competent software engineers assisted by automated, intelligent, tools in a friendly environment.

Like other major initiatives, such as the development and standardization of COBOL, the impact of STI will reach far beyond DoD. Indeed, whereas the standardization of COBOL was an excellent step at the time, it is now necessary to move forward to avoid outmoded practices and languages.

Other nations use modern methods to develop more usable systems. The Japanese Government, recognizing that software is critical to the success of their industry and that Japan currently lags behind the U.S., recently started a program to encourage software research and development. [See Electronics, Mar 27, 1980, pp. 113-136.] By inaction we may open a critical software gap between this country and

others that is just as severe as the gap in some industries between our aging industrial base and more modern plants in other nations.

Appendix C discusses current DoD software problems in more detail, and appendix D presents summaries of twelve important previous DoD studies in the field.

#### 1.1.2 Timeliness of the STI

There are a number of reasons for a Software Technology Initiative at this time. In the past few years a number of efforts have begun to attack the problems of software development and maintenance, and it is now widely believed that the field of software engineering is poised for significant consolidation, and possibly for a major breakthrough.

Now is an especially propitious time for the Initiative. The adoption of Ada as the standard language for military systems introduces a new software environment, as yet unpolluted by inadequately defined, poorly documented, hardware-dependent, or unstructured software. There are no large numbers of poorly trained Ada programmers. If the Software Initiative is successful, its ideas will engender software applications that are more reliable, less costly, more useful, and more timely than most software is today. If the well-documented military computer system failures of the past are not to be repeated during the next few years, it is imperative to act decisively to integrate and disseminate those advances that have been made piecemeal over the past few years, and fund the research and development that will result in a major improvement in software production and quality.

#### 1.1.3 History

DoD attempted to organize the field of software engineering research by running a conference in September, 1973, in Monterey, California, titled "The High Cost of Software." The conference was

sponsored by a Tri-Service Committee composed of representatives from the Office of Naval Research, the Air Force Office of Scientific Research, and the Army Research Office. The conference had the explicit objectives of "providing more precise description and measure to those problems of [software] cost, reliability, and error." [Research Directions in Software Technology, Peter Wegner, ed., MIT Press, 1979, p. xi] It was the hope of the sponsors that they "had initiated a process which would expedite the evolution of software development from the art form it largely is today to a discipline with scientific rigor and its own well-defined structures and formalisms." [Ibid.]

The conference did, indeed, start such a process. A "Software Management Steering Committee" was formed in December, 1974, at the request of the Director of Defense Research and Engineering, the Assistant Secretary of Defense for Installations and Logistics, and the Assistant Secretary of Defense for the Comptroller's Office. Representatives from all three services were included. The prime focus of the Committee was to be improvement of acquisition, development, and maintenance of weapons systems software.

Starting at the same time as the committee were two parallel research projects, one at the MITRE Corporation and the other at the Applied Physics Laboratory of the Johns Hopkins University. The purposes of these projects were

"to identify and define (1) the nature of the critical software problems facing DoD, (2) the principal factors contributing to the problems, (3) the high pay-off areas and alternatives available, and (4) the management instruments and policies that are needed to define and bound the functions, responsibilities and mission areas of weapon systems software management." [DoD Weapons System Software Management Study, A. Kossiakoff et al., Johns Hopkins Applied Physics Lab, June 1975, p. E-5]

The studies began in January, 1975, and were completed by that June. Because the allotted time was so short, the researchers were restricted to reviewing ten recent, major, DoD-sponsored studies of software development (usually "lessons learned"), reviewing the software design and management of a number of additional Army, Navy, and Air Force projects, and interviewing industry representatives. Nevertheless, the studies were excellent and provided a good foundation for later work.

The two studies were delivered in July, 1975. In March, 1976, the Software Management Steering Committee issued the Defense System Software Management Plan, which was followed in April by DoD Directive (DoDD) 5000.29, "Management of Computer Resources for Major Defense Systems." These documents were a start at integrating and directing the software research underway at DoD, and at modernizing the management of software projects.

Part of DoDD 5000.29 established the Management Steering Committee for Embedded Computer Resources, which then established the Research and Development Technology Panel. The Panel published the first Defense System Software R&D Technology Plan in September, 1977, to cover the period FY 1978 to FY 1983. "It provided for the first time a common structure for all DoD software R&D programs. For each of twelve technology areas, problems and issues were listed, existing and proposed R&D directions described, and recommended funding profiles presented." [Defense Computer Resources Technology Plan, Management Steering Committee for Embedded Computer Resources, USD (R&E), June 1979, p.1] Since then the plan been updated once: the second edition was published in June, 1979.

Meanwhile, other across-the-board studies were conducted in the Federal government, as the software problem became increasingly obvious. Some examples are the study done by the Software Acquisition and Development Working Group for the Assistant Secretary of Defense

for C3I, published in July, 1980, and the 1978 "Federal Data Processing Reorganization Study."

The concept of a Software Technology Initiative was first introduced by Dr. Ruth M. Davis, then the Deputy Under Secretary of Defense for Research and Advanced Technology (DUSD(R&AT)), in her testimony before the House Armed Services Subcommittee on Research and Development on 5 April 1979. The concept progressed towards realization over the last two years. The Report of the Secretary of Defense to Congress in January 1980 concerning FY81 and beyond stated, "In FY81 we will begin a major new initiative in computer software technology." Primary emphasis was planned for FY82-86. In FY80 the planning for the Initiative was assigned to the Office for Electronics and Physical Sciences, specifically to Dr. David Fisher.

After some preliminary activity, a DoD Workshop on Software Technology was held at Fort Belvoir, Virginia, in May, 1980. It was aimed at providing a common basis for planning the Initiative. The first day provided background presentations, and the second day consisted of four parallel working group sessions on the topics: High Pay-Off Thrusts, Planning Strategy for the Initiative, Mission Area Orientation, and Technology Transfer. The workshop produced a basis for the structure of the Initiative and a set of preliminary goals and thrusts.

The Software Technology Coordinating Committee, formed to provide direction and oversight to the Initiative, made some progress during its meetings over the following summer. A draft of a management-oriented Program Definition Plan was distributed for comment in July, 1980, and an announcement appeared in Commerce Business Daily on 30 July asking for qualification statements by 31 August.

Over eighty qualification statements were received and reviewed. In an effort to reach a wider audience, especially the academic and private research communities, the announcement was repeated in the

December, 1980, Communications of the ACM. Approximately forty additional replies were received.

During the fall of 1980, comments were collected on the July draft of the Program Definition Plan. It became clear that a major difficulty was the need for more technical substance in the Program Definition Plan and in STI planning in general. As a first step, work began to identify potential technical thrusts and to compile what was tentatively called a "Technical Development Plan." The Technical Development Plan evolved into this document, the main purpose of which is to begin the process of choosing specific initial R&D thrusts for the Initiative.

#### 1.1.4 Relationship to Other Efforts

The STI was originally visualized as falling between basic research aimed at long-term payoffs, such as might be supported by DARPA and the National Science Foundation; and the more short range efforts that comprise most of current DoD software R&D. In DoD parlance, it was to be "6.2 and 6.3 moneys rather than 6.1." Payoffs were to be obtained by ensuring that the results would be used by the Services in their systems; therefore, the results would have to be practical, cost-effective, and effectively transferred to the users. Although the details of the Initiative are still evolving, the original program's intent remains unchanged.

R&D efforts under the Initiative will be performed through the same DoD organizations that currently administer such efforts, but with overall management from OUSDRE. To be effective, the STI must be funded substantially above current levels of DoD software R&D funding. Realistically, however, the extent of additional funding will depend on the quality of the proposed program and the expected impact of the results.

The Software Technology Initiative will be conducted under the auspices of the Research and Development Technology Panel of the

Management Steering Committee for Embedded Computer Resources. Coordination will be close with the Ada Joint Program Office, with DARPA, with the other DoD Agencies concerned with software development, and with the military services and their laboratories. Liaison will be close with the Very High Speed Integrated Circuit (VHSIC) Program to provide coordinated and integrated thrusts where appropriate. Coordination will also occur with the DoD Computer Security Initiative. Plans are being formulated to establish a formal mechanism to obtain inputs from the industrial and academic communities in the subject areas of information sciences and software technology. Other ad hoc activities will also be used to collect inputs from a number of different perspectives.

The details and boundaries of the Software Technology Initiative are still being defined. A revision of the Program Definition Plan will address the organizational and managerial issues. This document primarily addresses the technical aspects of the program.

## 1.2 Definitions

A thrust is an effort to provide or apply a technical capability. A thrust may involve hardware, software, documentation, procedure, management, or education; many thrusts involve combinations of these. This document deals with candidates for thrusts. Other topics, especially those furthest from implementation, need additional research to define possible concrete results.

Candidates are categorized according to the period (after initial funding) when their benefits will begin to be realized throughout DoD. Benefits from short-term candidates should be realized within 4 years, medium-term candidates within 7 years, and long-term candidates after 7 years. The estimate for each thrust is measured from its starting time, and assumes that research and development are supported at appropriate levels through the thrust's lifecycle. Almost all thrusts produce preliminary results; success-

ful thrusts continue to produce results until superseded or subsumed by other thrusts' results.

### 1.3 Themes

Productivity, reliability and maintainability, and technology transfer are the primary goals of the Software Initiative. Improved training and automated support increase productivity, thereby alleviating the critical shortage of software personnel. More complete and less ambiguous requirements, more accurate specification, better languages and translators, software verification, and improved testing techniques are essential to increased software reliability (therefore, system reliability). Modular software, better documentation, complete capture of project history, and methodologies allowing for change are needed to permit new requirements to be implemented and to take advantage of new technology. Technology transfer, without which technological innovation is meaningless, requires training for software personnel, managers, and contract monitors. They need to understand the benefits from the use of new tools and procedures.

Standardization, complexity, and human cognitive limitations are key issues. A standard programming language, a standard operating system interface, standard practices, and standard tools can make software interoperable and people interchangeable, but standardization can stultify innovation. System complexity can be reduced by partitioning problems into smaller problems with clean interfaces; however, there will always be problems whose decompositions are unknown. Problems for humans to solve need to be scaled to human cognitive abilities. The need for concepts to be packaged so that only their behavior matters is obvious. Consider how little needs to be known about a complex computer and its operating system before the computer can be used effectively. These thoughts motivate the selection of thrusts for the Software Technology Initiative.

#### 1.4 Candidate Lifecycle

The individual activities will vary among the candidates because of the diversity of products (software, hardware, training materials, management practices, regulation revisions, etc.), and variations in the beginning level of understanding (identified symptoms, understood problems, known solution approach, existing prototype, etc.). Despite their differences most will follow the general pattern in Figure 2.

In well understood areas, the Problem Definition and Research stages may be bypassed. In other areas, problem definition, exploratory research, and evaluation of potential and feasibility may need to be performed. Research prototypes will be built and tested. Where prototypes or products already exist, they may be used in the experimentation.

The Planning stage prepares for the following stages. In Development, production prototypes lead to more knowledge of requirements, and operational integration and performance problems. Development of candidate products will require careful engineering and management, and will usually pass through the normal development stages for the types of products involved.

For each product, either in prototype form or after development, a formal evaluation needs to be performed to establish and--to the extent possible--quantify the improvements provided. The evaluation may also lead to revisions.

A technology transfer strategy needs to be established early, and packaging and auxiliary products or capabilities for technology transfer developed. These might include training materials, instructors, publicity, coordination with other candidates or efforts, arrangements for pilot demonstrations, portability and installation aids, and regulation and standards revisions. Substantial efforts

- STAGE 1: PROBLEM DEFINITION
- Agree on Thrust Area
  - Detail Problem
  - Develop Detailed Research Plan and Preliminary Development, Evaluation, Technology Transfer, and Maintenance Plans
  - Issue RFP for Research
  - Award Research Contract(s)
- STAGE 2: RESEARCH
- Identify/Generate Solution Approaches
  - Design and Implement Research Prototypes
  - Experiment with Evolving Prototypes
  - Evaluate Results
  - Prepare Requirements Description
- STAGE 3: PLANNING
- Plan and Coordinate Technology Transfer Strategy
  - Develop Detailed Plans for Development, Evaluation, and Technology Transfer
  - Update Preliminary Plan for Maintenance
  - Issue RFP
  - Award Contract
- STAGE 4: DEVELOPMENT (INCLUDING TECHNOLOGY TRANSFER ITEMS)
- Production Prototype
  - Design
  - Construct
  - Verify
- STAGE 5: EVALUATION
- Demonstrate and Measure in Field
  - Evaluate Results
  - Revise if Required
- STAGE 6: TECHNOLOGY TRANSFER
- Outreach within DoD
  - Assist in Introduction
  - Follow-up
- STAGE 7: MAINTENANCE
- Transfer to Maintaining Organization
  - Incorporate/Replace with Improvements

Figure 2: Typical Thrust Lifecycle

may be required to launch a developed capability and to achieve its widespread and continuing usage.

The developed products will become the responsibility of some organization to distribute, maintain, and improve. In cases where the STI is continuing to work in an area towards even more advanced capabilities, improvements may be an STI responsibility. At each stage, and within some stages, there will be a need for DoD to make decisions on the continuation, direction, and funding level of a candidate.

### 1.5 Document Organization

The following three chapters contain descriptions of research and development candidates for the Software Technology Initiative identified from a study of the professional literature, the opinions of experts in the field, and responses from industry and research establishments to DoD's requests for information [Commerce Business Daily (July 30, 1980), Communications of the ACM (December 1980)]. Within each chapter candidate thrust areas are grouped according to their relevance to technical, managerial, personnel-related, and continuity-related problems. Chapter 5 summarizes Chapters 2 through 4 and the goals of the STI.

Detailed descriptions of candidate thrusts are to be found in Appendix A. Appendix B contains short descriptions of topics excluded from the list of candidates because they could not be associated with specific capabilities. Appendix C defines software problem areas, and Appendix D summarizes earlier efforts to define and attack the software problem. Appendix E is a reviewer's guide to evaluating and selecting thrusts. A reviewer's response questionnaire is in Appendix F. Reviewers are asked to use the questionnaire to comment on, evaluate, and rank the candidate thrusts.

## 2.0 THRUSTS WITH SHORT-TERM PAYOFFS (UNDER 4 YEARS)

### 2.1 Introduction

Short-term thrusts will obtain or expand capabilities already in operational use within commercial or government organizations (including elements of DoD). These capabilities may need to be completed and re-packaged, but do not require further research. The general term for this process is Technology Transfer, which means putting already known technology to work throughout DoD.

There exist a number of techniques and practices generally acknowledged to be beneficial to the software process that do not need further R&D, but rather internal DoD efforts to ensure that these techniques are being utilized to the greatest possible extent. A technology transfer thrust will produce the necessary implementation plans, support tools, and training materials. Structured programming, avoidance of assembly language, avoidance of vendor-dependent extensions of standard languages, modularization of functions, accurate documentation creation, and monitoring by management of programmers' adherence to standards are all immediately available, beneficial concepts to be exploited. In addition, there are a number of medium- and long-term thrusts that are structured to produce intermediate, short-term benefits.

### 2.2 Candidates

The tentative set of candidates with significant short-term payoffs is shown in Figure 3; detailed descriptions are in Appendix A. The remainder of this chapter discusses the short-term thrusts in general terms.

It is postulated that small groups of highly competent software engineers produce better systems than large groups of mediocre programmers. If this can be demonstrated, it will be useful to have a

- Technical
  - General (Across Lifecycle)
    - Integrated Software Support Environment
    - Ada Package Sets for Common Usage Areas
    - System Dictionary/Directory
    - Programmer Workstation
    - Useful Measures of Software Quality
  - Concept/Feasibility
    - Rapid Simulation
  - Requirements
    - Rapid Prototyping
  - Design & Programming
    - Predicate Approach
  - Testing
    - High Confidence Software Testing
  - Operations
    - Impact Analysis of Proposed Change
- Managerial
  - Software Technology-Compatible Acquisition
  - Technology Transfer
- Personnel-Related
  - Superperformer Competencies
  - Improved Education About Software
- Continuity-Related

Figure 3: Short-term Candidates

mechanism to identify competent and potentially superior programmers, and to provide them with productivity-increasing tools. Several thrusts are related to this thought.

#### 2.2.1 Technical

Software engineers need the support of good tools. An important thrust would be the development of an Integrated Software Support Environment (ISSE) for compatible tools to facilitate software creation, debugging, modification, and documentation. Editors, language translators, change-level recording file systems, system dictionaries and directories, and data bases of standard software modules are some of the software components of an ISSE. As the set of tools expands to cover the entire project lifecycle, the ISSE will evolve into a software management environment in which a project's entire history is captured.

The basic hardware component of the ISSE will be a workstation incorporating multi-media input and output devices. (The usefulness of color, graphics, multiple displays, and voice need to be investigated.) Workstations will be connected to a system that permits coordination of the efforts of several groups working on a project. This system will be a repository for implementation notes, convention descriptions, common modules, and software that exceeds the workstation's capabilities. Networks of processors could become operational within the short-term time frame. Eventually, as the products of the Initiative become available, interconnected workstations and processors will have access to local and distant resources: data bases, computer peripherals, and expert advice via conferencing.

Measures of software quality are needed, as well as tools to evaluate the quality of software products. A thrust could produce measures of software quality and tools to evaluate the quality of software products. The results could help to evaluate programmers, choose among several alternative packages, provide incentives, and

ensure minimally acceptable quality levels for contractor-supplied software.

Simulation of the major functions of a system will lead to quicker feasibility determination and better performance requirements. This is especially significant for subsystems that contain separate processors, in multi-computer networks, because of the large number of available options and the novelty of these designs. More rapid, more accurate simulation languages and techniques need to be developed.

Another potential thrust is the development of a capability to produce prototype versions of variations of embedded systems rapidly. A prototype exhibits the functional behavior of the target system (perhaps a subset in a simulated environment), though perhaps not its real-time characteristics. Prototyping is especially useful for those parts of systems that interact with people, because it provides feedback to the system designers on the suitability of the interface, and also because it gives the system's future users valuable early experience. Recent progress in rapid prototyping for ADP applications engenders optimism that the practice can be transferred to the embedded systems area.

The possibility of including assertions within a program about its status at various points--sometimes called the predicate approach--has been discussed for over ten years. Although no compiler prevents a programmer from including code to accomplish this--and even to compile it conditionally on some switch--it is rarely done. A formal effort to require assertion predicates at critical points (with or without new language constructs) should result in more reliable software, as more logic errors are detected during the software development stage. Also, experience with the use of assertion predicates will assist efforts in automatic software verification.

Recent theoretical and practical advances in test data design could be applied to DoD use. Software tools to record the scope and sophistication of the coverage of a program's logic paths by tests, to facilitate early testing of system components, and to analyze test results could become operational in this time frame. Better test data combined with better tools will significantly increase DoD's confidence in tested software.

Software that is used regularly changes; at least, the demand for change increases. The benefits of changes need to be weighed against the costs, but the costs may be difficult to measure without extensive investigation, because there currently is no accurate way to know how much software is affected. Means to measure of the full impact of changes will require many years to develop; however, tools are available to help programmers see what modules have to be looked at when certain kinds of changes are proposed. The usefulness and possible enhancement of these tools need to be investigated.

#### 2.2.2 Managerial

Programmers are not alone in needing support. System acquisition managers and program managers need to be more aware of the realities of software development: what realistically can be expected, when, and at what cost. Those responsible for defining software requirements need to be up-to-date on evolving possibilities for hardware and software. Procurement standards need to be developed so that software obtained from contractors meets specified standards of quality, production, and format. Managers need metrics and aids to evaluate contractors' performance. Thrusts in the management area will produce needed tools and training materials.

#### 2.2.3 Personnel-Related

Competencies and attributes of good designers, analysts, and programmers need to be identified so that potentially superior personnel can be selected from candidates or produced through training.

(Potentially inferior personnel also need to be identified.) Aptitude criteria need to be developed and refined. The technical and organizational skills needed to work on large projects need to be defined and taught effectively. Schools could be encouraged to try out and evaluate new training techniques.

The talents of superior software personnel could be better organized to increase their collective productivity. Instead of dispersing the best people among various projects, it has been suggested that they function in groups to define and structure projects for others to expand and implement. A thrust could examine this possibility, test it out on a pilot project, and determine its feasibility.

#### 2.2.4 Continuity-Related

In the short term, the problems of maintaining project continuity through staff and requirements changes seem to be matters for project management, rather than R&D thrusts. Until more automated tools are developed for capturing the history of a project, managers will have to ensure that documentation is kept current, that teams take responsibility for team members' contributions, and that idiosyncratic practices are either prohibited or completely documented. The ways people are used, treated, and managed could be improved with a corresponding improvement in this area.

### 2.3 Chapter Summary

The specific thrusts that could lead to the capabilities discussed above are described in more detail in Appendix A. Many of these projects are ambitious, and some will yield only preliminary capabilities in the short-term time frame. Although it is important to produce useful results and to establish momentum, it is also important not to settle only for what now seems readily attainable. These pioneering thrusts will set the tone of the Initiative.

### 3.0 CANDIDATES WITH MEDIUM-TERM PAYOFFS (4 - 7 YEARS)

#### 3.1 Introduction

Medium-term candidates are designed for areas where preliminary work indicates that major results may be expected in a few years. Prototypes should be ready for field testing and use in the late 1980's. Many short-term thrusts are continued through the medium term, because they will continue to produce meaningful results. New medium-term candidates include those that depend on results from the short-term thrusts (e.g. transforming software to improve quality) and those that have a long gestation period (e.g. forgiving systems).

#### 3.2 Candidates

The medium-term candidates are listed in Figure 4. The remainder of this chapter discusses the medium-term thrusts in general terms; more complete descriptions of the individual thrusts appear in Appendix A.

##### 3.2.1 Technical

Work on the Integrated Software Support Environment would continue. The overall design would be completed, providing the framework into which support tools (packages, etc.) would be integrated. Results from thrusts dealing with the creation of compatible tools covering the entire software lifecycle, thrusts to create standard Ada package sets, and a thrust to create a system directory/dictionary enhance the ISSE in the medium term. These tools will ensure that development data are compatible throughout a software system's lifecycle. The work on programmer workstations and on measures of software quality would continue; early results from these two thrusts would be augmented as experience is accumulated.

A thrust to investigate multiple representations of software (different notations and different media) would show results in this

- Technical
  - General (Across Lifecycle)
    - Integrated Software Support Environment
    - Ada Package Sets for Common Usage Areas
    - System Dictionary/Directory
    - \*Sets of Tools Covering Entire Lifecycle
    - Programmer Workstation
    - Quantitative Measures of Software Quality
    - \*Multiple Representations of Software
    - \*Configuration Independence
  - Concept/Feasibility
    - Rapid Simulation
  - Requirements
    - Rapid Prototyping
    - \*Application Domain Expertise
    - \*Data Validation
    - \*Built-in Testing
    - \*Forgiving Systems
  - Design
    - Predicate Approach
    - \*Data Flow
    - \*Exception Handling
  - Programming
    - (Predicate Approach)
    - \*Transform Software to Improve Quality
  - Testing
    - High Confidence Software Testing
  - Operations
    - \*Facilitating System Evolution
    - (Impact Analysis of Proposed Change subsumed)
- Managerial
  - Software Technology-Compatible Acquisition
  - Technology Transfer
  - \*Acquisition Manager's Support System
- Personnel-Related
  - Superperformer Competencies
  - \*Programmer Laboratory
  - Improved Education About Software
  - \*Personnel Independence
  - \*Intensive Advanced Programmer Training
- Continuity-Related
  - \*Voice Replaces Text
  - \*Built-in Training and Documentation

(\* indicates new this period)

Figure 4: Medium-term Candidates

time period, as would work on configuration independence (system operability in different hardware and operating system environments). Standardized internal boundaries between modules and layered design will facilitate software system modifications.

Work on rapid simulation and rapid prototyping will continue. Study of DoD application domains would reveal areas where program commonalities can be exploited. Work on data integrity would create better methods for dealing with incomplete or corrupted data. Built-in testing would improve system dependability and availability, because software would incorporate advances in methodologies for self-testing before and during operation. Work on forgiving systems, tying in closely with other thrusts, would create systems more forgiving of errors made by system components or by humans. A parallel thrust on exception handling will support the work in forgiving systems. The result of these thrusts will be more robust systems, less likely to fail in the event of problems with input data, the environment, or their own behavior. [For example, failure to provide for synchronization of the back-up computer with the on-line computers delayed the initial Space Shuttle launch.]

As a consequence of the rapid decline in digital hardware costs, the desire for distributed applications, and improvements in the capabilities of microprocessors, high-performance embedded systems will be based on multiple processor designs. A thrust is needed to develop the expertise to use these designs effectively. Current computer languages may not be adequate to describe the processes running on such systems; therefore, investigation of new process representations, such as the data flow approach (processes described by data transformations, rather than control sequences), are necessary. Results of the data flow investigation in the 4 to 7 year time period will facilitate software creation for these new architectures.

Another new thrust is tied into the short-term thrust to create measures of software quality. As that thrust continues, it will become easier to reward producers of good software. In addition, it may be possible to create software packages to transform existing programs into forms with better software quality scores. The transformation package will make systems easier to comprehend and maintain, and it might also find previously undetected errors in logic. This capability will benefit existing systems, as well as new systems.

Knowledge obtained from the short-term thrust dealing with impact analysis of proposed changes will be incorporated into a thrust that will examine the evolution of large-scale systems, looking for methods for making changes to such systems (especially unanticipated changes) easier.

#### 3.2.2 Managerial

Thrusts dealing with technology transfer and the acquisition of software will continue in the medium term. The results of these thrusts will form the base for the development of an automated acquisition manager's support system: a knowledge-based system used by managers to prepare contractual documents, deal with contractors, and monitor projects.

#### 3.2.3 Personnel-Related

Because of the rapid turnover in personnel and the problems one person has in taking over from another, due to faulty code or inadequate documentation, it is necessary to investigate methods for improving and standardizing software specialists' work. It is also necessary to investigate methods for reducing turnover and its impact. Thrusts whose products capture project history, enforce standards, and implement standard software support environments will help to ease these problems.

Educational materials to train persons responsible for software specification will be produced, and more effective means of identifying the qualities of superior software personnel will come into use. A laboratory to study the behavior of software personnel could identify specific good and bad habits, and try out new techniques in a controlled environment. New tools and procedures resulting from these experiments will make it easier (faster, less expensive) to train better software designers, analysts, and programmers.

#### 3.2.4 Continuity-Related

Two thrusts in communication show promise of results in the medium term. The first, the use of voice input in addition to text, will make capturing information easier, if staff personnel prefer dictating or other verbal devices to typing.

A greater impact in the area of communications may be obtained from a thrust towards the use of built-in training and documentation. Training aids and documentation built into systems permit users and maintainers to learn through use without fear of destroying the system in the process. On-line documentation and tutorial, step-by-step instructions help assure accurate understanding on the part of users and maintainers.

### 3.3 Chapter Summary

This period could witness changes in the software development process: automated tools could supplant today's manual and semi-automated procedures; small groups of software engineers assisted by these tools could do the work of today's large groups of programmers and analysts; and software products could be more reliable and timely, and more closely resemble real requirements.

The medium-term candidates emphasize the areas of software tools, personnel improvement, and management. Because many of them are interrelated, there will be a synergistic effect: success in some

will enhance the value of success in others. To assure this, integration of the tools and other products needs to be a high priority concern.

#### 4.0 LONG-TERM CANDIDATES (MORE THAN 7 YEARS)

##### 4.1 Introduction

Long-term candidates are directed at problems without currently available solutions. These problems are:

- o how to build reliable, intelligent systems to augment human decision-making (knowledge-based systems),
- o how to use networks of large numbers of special- and general-purpose processors effectively,
- o how to integrate existing tools so that their users are conscious of a single system.

The following sections discuss the identified long-term thrusts, whose major impact is not expected to be felt before 1990. Break-throughs in any of these areas might, of course, produce dramatic results sooner.

##### 4.2 Candidates

The long-term thrusts are summarized in Figure 5. More complete descriptions of long-term candidates are included in Appendix A. Many long-term candidates are expected to provide useful preliminary capabilities in the earlier years of the Initiative, as discussed in the preceding chapters. Even after the primary efforts have been completed, continual monitoring, evaluation, and refinement of products from earlier phases of the Initiative will be required until products are absorbed or superseded by new products.

###### 4.2.1 Technical

Intelligent (knowledge-based, "expert") systems could assist software engineers by providing application domain knowledge, by automatically handling low-level coding chores, by checking for

- Technical
  - General (Across Lifecycle)
    - Integrated Software Support Environment
    - \*Software Engineer's Support System
    - \*Earliest Possible Detection of Errors
  - Concept/Feasibility
    - Rapid Simulations
  - Requirements
    - Rapid Prototyping
    - Application Domain Expertise
    - Data Validation
    - Forgiving Systems
    - \*User Oriented Requirements Interfaces
    - \*Complex Knowledge-based Systems
  - Design
    - Data Flow
    - Predicate Approach
    - Exception Handling
    - \*Self-interfacing Software
    - \*Distributed Functions and Resources
    - \*Suitable Communication Interconnection
  - Programming
    - Transform Software to Improve Quality
    - \*Formal Verification of Large Systems
  - Testing
    - {None}
  - Operations
    - Facilitating System Evolution
- Managerial
- Personnel-Related
  - Intensive Advanced Programmer Training
  - Personnel Independence
  - \*User Programming
- Continuity-Related
  - Built-in Training and Documentation

(\* indicates new this period)

Figure 5: Long-term Candidates

common programming pitfalls, by verifying assertions, by locating relevant standard modules, and by coordinating the efforts of several design teams. (The possibilities are endless.) Ultimately there could be one "expert", which uses various application-domain data bases, human-interaction data bases, and individual preference data bases, as needed, to interact with systems and users. The structures of these data bases are unknown, as is the magnitude of the effort required to create them and maintain them in perpetuity. Knowledge-based systems will be integral parts of the evolving Integrated Software Support Environment that multiplies the productivity of software engineers.

Knowledge-based simulation systems will help designers construct more accurate simulations. These systems will use knowledge of available hardware and software modules, standardized techniques, and extracts from simulations of related applications. Analogous systems will assist in the construction of prototypes and in the evaluation of prototype results.

More widely available education about computers and software engineering will gradually increase the sophistication with which users determine and define their computing needs. Automated requirements languages will help persons responsible for system specification to define complete and unambiguous requirements for new systems. Knowledge of costs, capabilities, and good industry practices will need to be incorporated into these systems.

Design techniques proven successful in earlier phases of the Initiative will be refined. Data flow techniques may evolve into data-directed design and programming languages. Translators for these languages could generate modules for execution in distributed computing systems, using special-purpose hardware, when appropriate. Languages could support the incorporation of assertions, perhaps assisted by systems that recommend places for assertions, or even add

and check them, automatically. Standard, parameterized components may be available; component-tailoring systems could connect components by appropriate interface mappings.

More effective approaches for dealing with real-time problems, such as interrupts, data exceptions, faulty hardware, and inter-process communication and coordination will become available as a result of experience with various alternatives in earlier Initiative phases. Formal verification of the correctness (i.e. conformance to specifications) of programs (especially, standard modules) will be better understood. Metrics for software quality may lead to systems capable of reducing complexity. (It's possible to envisage an interactive system suggesting alternative constructions to the software engineer's first attempts.)

#### 4.2.2 Managerial

The management of the software process will be more automated, with most of the decisions captured for analysis by systems that look for potential sources of trouble. Reports of project progress and adherence to schedules will be more timely and more accurate when this data is available in the computerized system.

#### 4.2.3 Personnel-Related

Automated assistance will multiply the productivity of software engineers so that fewer people are needed for projects of comparable size and complexity to today's projects. In some areas, systems will progress to the point where users construct programs themselves in cookbook fashion, or in a three-way collaboration among users, software engineers, and intelligent support systems. Advanced training tools and techniques, systems that forgive mistakes and point out the correct approach, and good human engineering of systems will decrease the cost of training the next decade's software practitioners.

#### 4.2.4 Continuity-Related

Software projects have to survive changes in personnel. Procedures must be developed through which replacement personnel can be brought up-to-date soon after assignment to a project. On-line diaries, change histories, accurate documentation, and automated support systems will help. The problem is to capture and then to organize all pertinent project data. The capabilities to accomplish this will evolve continuously.

#### 4.3 Chapter Summary

The view of the software world of 1990 and beyond, described in this chapter, is certainly optimistic, and probably sounds fantastic. However, there is no more reason to suppose a limit to the capabilities of computer-based systems than there is a reason to suppose a limit to human capabilities; both will evolve continuously. It is unlikely that more than rudimentary successes in developing intelligent systems will be achieved over the next ten years. It may be that results will always seem rudimentary as human capabilities increase, applications become more complex, and new problems come into focus.

Solution of the computing problems of the future will depend on effective use of knowledge-based systems, realization of the potential of multi-computer systems, and integration of tools provided by the Software Initiative into a coherent system.

## 5.0 SUMMARY

The preceding three chapters discussed thrust areas and candidate technical thrusts. In the short term, the goal is to acquire existing capabilities for DoD use. The medium-term goals are the improvement of capabilities already acquired and the obtainment of additional capabilities to fill in gaps, with emphasis on the integration of capabilities. In the long term, the goal is continued development, evolution, and integration of advanced capabilities.

Short-term benefits will come mainly from technology transfer, early results of thrusts with principal payoffs in later periods, and standardization efforts. The technology to be transferred already exists, at least in prototype. One important exception involves identification of superperformer characteristics and practices. Short-term results from research in this area will be applied to practice as rapidly as possible. Preliminary results are also expected in such areas as rapid prototyping and rapid simulation. Standardization efforts include the integrated software support environment (ISSE), programmer workstations, and standard Ada packages.

For the medium-term, the emphasis is on completeness and integration of tools, and on personnel and management issues. Complete sets of compatible tools will be developed, and capabilities will be added to the ISSE. Personnel training and organization, along with improved acquisition management support, will have significant payoffs.

In the long-term there is hope for synthesis of techniques, effective use of multiprocessor systems, and the incorporation of application domain expertise into systems. Automated software development expertise and application domain expertise will be available. Advances in digital hardware design and decreasing hardware costs will provide the computing power for the automation of func-

tions now performed manually by software personnel, and for new functions.

The underlying themes in Chapter 1 persist: the goals of productivity, reliability and maintainability, and technology transfer; and the issues of standardization, complexity, and human cognitive limitations. The technology transfer and human engineering aspects of the Initiative need to receive early attention and major emphasis, because the benefits of technology will be lost unless organizations are willing to acquire its products and people are willing (and able) to use them. Technical products must be designed and presented so that people see a direct benefit to their work assignments from their use of the products.

We have some understanding of the software problem and a substantial number of ideas towards its solution. Although there is no single obvious path to order-of-magnitude improvement, the technical basis can be organized for an effective Initiative that will increase the utility and reduce the expense, time, and risk of software projects. With appropriate funding and support, the Software Technology Initiative will lead the way to major improvements in productivity and software quality.

## A. DESCRIPTION OF CANDIDATES

This appendix contains individual descriptions of the thrust candidates mentioned in Chapters 2 through 4. The intent is to provide basic information for each candidate to help reviewers evaluate its potential contribution and its appropriateness for selection as an STI thrust. Reviewers may want to complete the questionnaire (Appendix F) while reading this appendix. The format for each candidate is:

Description. The central concept or goal of the thrust, the motivation for including it, its relationships with other candidates, and its benefits are briefly described. Benefits are entirely qualitative, but mention some of the concerns noted in Appendix E. Narrowly defined candidates and those aimed at enhanced utility may have low ratings on expected cost savings.

Some Relevant Research and Products. This section gives a general indication of the state of the art for the idea. No attempt at completeness is made, but notification of any major omissions is welcomed.

Remarks on Rationale. This section is included for some candidates where additional explanation of the underlying motivation is thought to be needed.

References. The references are entry points to the literature or recent examples. They are not exhaustive, but often a reference will have its own extensive bibliography.

Figure 6 lists the titles of candidates in Appendix A, and Other Ideas, which are in Appendix B. The titles are grouped under the same headings used in Chapters 2, 3, and 4, as an aid to the reviewer. Page numbers for each candidate and "other idea" are in the Table of Contents.

<u>Candidate</u>	Appendix A			Appendix B
	Payoff Periods			<u>Other Ideas</u>
	<u>Short- term</u>	<u>Medium- term</u>	<u>Long- term</u>	
Technical				
General				
Integrated Software Support Environment	X	X	X	X
Presentation and Manipulation				
Ada Package Sets for Common Usage Areas	X	X		
System Dictionary/Directory	X	X		
Set(s) of Tools Covering Entire Lifecycle		X		
Software Engineer's Support System			X	
Programmer Workstation	X	X		
Useful Measures of Software Quality	X	X		
Multiple Representations of Software		X		X
Rigorous Documentation				
Conflict Recognition among Representations				X
Earliest Possible Error Detection			X	
Exploratory Systems Applications of VHSIC				X
Military Information Utility				X
Multiple Classes of Service		X		X
Configuration Independence				
Standard Real-Time Operating System				X
Conception/Feasibility				
Rapid Simulation	X	X	X	
Requirements				
Rapid Prototyping	X	X	X	
Application Domain Expertise		X	X	
Data Validation		X	X	
Built-in Testing		X		
Forgiving Systems		X	X	
User-Oriented Requirements Interfaces			X	
Complex Knowledge-Based Systems			X	
Rapid Derivation of Requirements				X

Figure 6: Summary of Candidates' Tentative Status

<u>Candidate</u>	Appendix A			Appendix B
	Payoff Periods			<u>Other Ideas</u>
	<u>Short- term</u>	<u>Medium- term</u>	<u>Long- term</u>	
Transform Informal to Formal Requirements				X
Requirements Languages Transformation				X
Weakest Possible Requirements Description				X
Design				
Derivation of Software from Specifications				X
Data Flow Approach		X	X	
Self Interfacing Software			X	
Predicate Approach	X	X	X	
Very High Level Languages				X
Exception Handling		X	X	
Component Tailoring and Interfacing				X
Publication of Standard Designs				X
Data Structure and Abstraction				X
Distributed Functions				
Distributed Functions and Resources			X	
Suitable Communication Interconnection			X	
Programming				
Code Skeletons				X
Graph-Oriented Language				X
Generating Assertions from Requirements				X
Transform Software to Improve Quality		X	X	
Transform to Satisfy Physical Constraints				X
Man-Machine Quality Improvement Team				X
Formal Verification of Large Systems			X	
Application Generators				X
Reusable Software				X
Actor Languages				X
Testing				
High Confidence Software Testing	X	X		
Static Analysis of Software				X

Figure 6: Summary of Candidates' Tentative Status (cont'd)

<u>Candidate</u>	Appendix A			Appendix B
	Payoff Periods			<u>Other</u> <u>Ideas</u>
	<u>Short-</u> <u>term</u>	<u>Medium-</u> <u>term</u>	<u>Long-</u> <u>term</u>	
Generating Test Data from Require- ments				X
Generating Test Data to Violate Assertions				X
Test Bed Facilities				X
Operations				
Facilitating System Evolution		X	X	
Construction for Future Evolu- tion				X
Modification of Large Systems				X
Impact Analysis of Proposed Change	X			
Management				
Acquisition Manager's Support System		X		
Software Technology-Compatible Acquisition	X	X		
Model Contracts for Buying Software				X
Technology Transfer in Software area	X	X		
Maximizing DoD Rights to Software				X
Multiplying Expert Effectiveness				X
Quick Look Feasibility/Evaluation				X
Personnel				
Superperformer Competencies	X	X		
Intensive Advanced Programmer Training		X	X	
Programmer Laboratory		X		
Personnel Independence		X	X	
Improved Education About Software	X	X		
User Programming			X	
Communication/Continuity				
Completely Captured Software				X
Multi-person Machine Mediated Programming				X
Totally Visible Software				X
Systems that Never Forget				X
Voice Replaces Text		X		
Built-in Training and Documentation		X	X	

Figure 6: Summary of Candidates' Tentative Status (concluded)

## A.1 Technical

### A.1.1 General

#### A.1.1.1 Integrated Software Support Environment.

##### Description

Emphasis here is on consistent tool interfaces and expandability of the environment and tool set. Potential users include programmers, analysts, designers, testers, and managers. The idea is to provide growing support for all phases of software development and maintenance.

A framework will provide compatibility and synergy among tools and among workers, and will anticipate open-ended improvement and growth. High levels of availability, reliability, and correctness will be needed for acceptance by software workers and for the development and maintenance of high quality software.

The integrated software support environment will evolve from or incorporate the Ada Programming Support Environment. The environment needs ease of learning and use, good visibility, effective communication among users, usage statistics to allow evaluation of environment elements, support for all DoD approved high order languages, support for multiple versions of software, portability, and the ability to support itself. Canonical representation forms will be standardized to aid in compatibility among tool sets.

Human interfaces require excellent human factors, with flexibility for growth and for use on varying hardware configurations. The interfaces may need to vary with task types and with individual approaches to problem solving.

This thrust is related to the ideas of a set of compatible tools covering the entire lifecycle, a software engineer's support system, a system dictionary/directory, completely captured software, multi-person machine mediated programming, totally visible software, systems that never forget, a programmer workstation, an acquisition manager's support system, multiple representations of software, and other thrusts resulting in software tools. This thrust provides a framework for products of other thrusts, and so increases their synergy, ease of use, and technology transfer.

The greatest improvement potential exists in synergy among tools and in cumulative improvement in productivity aids. Other benefits, including the minimalization of retraining requirements, will derive from standardization. Of course, the tools and facilities offered within the integrated software support environment will provide substantial benefits themselves, and it is difficult to differentiate their contribution from that of the framework. One might, however, estimate cost savings as an enhancement of some reasonable percent of the benefit from the remainder of the Initiative.

The benefits of this thrust have potential for the entire industry including all of the Federal Government. If the proper balance can be struck between standardization/compatibility and growth/innovation, the environment will continue to contribute for many years.

#### Some Relevant Research and Products

The Unix Programmer's Workbench (PWB) and Maestro are examples of commercially available programming support environments; active efforts are underway to design and implement the APSE. Software tools and software support systems are active areas of R&D. Most of the responses to the July 1980 CBD announcement mentioned tools, usually specific, isolated ones.

### Remarks on Rationale

A layer of structure is needed between the APSE and sets of individual tools. The layer would support languages other than Ada and provide standards for combined/hidden invocation, data base structures, management of multiple representations, etc. Within this environment, different sets of tools based on different methodologies can evolve, and yet preserve a significant interoperability.

### References

Buxton, J. N. "An Informal Bibliography on Programming Support Environments." ACM Sigplan Notices vol. 15, no. 12 (Dec. 1980), pp. 17-30.

Buxton, J. N., and V. Stenning. Requirements for Ada Programming Support Environments ("Stoneman"). Arlington, VA: DARPA, Feb. 1980.

Graham, R. M., chairman. "Proceedings of the ACM-SIGPLAN Symposium on the Ada Programming Language." ACM Sigplan Notices vol. 15, no. 11 (Nov. 1980).

Hausen, H. L., and M. Mullerburg. "Conspectus of Software Engineering Environments." In Fifth International Conference on Software Engineering, pp. 34-43. Los Alamitos, CA: IEEE Computer Society, Mar. 1981.

Riddle, W. E., and R. E. Fairley. Software Development Tools. Berlin: Springer-Verlag, 1980.

Wasserman, A. I., ed. "Special Issue on Automated Development Environments." Computer vol. 14, no. 4 (Apr. 1981), pp. 7-54.

#### A.1.1.2 Ada Package Sets for Common Usage Areas.

##### Description

One of Ada's major advantages is that it permits the development and use of standard software packages. Such centrally-maintained package sets would avoid effort duplication, yielding higher-quality software. Standard packages will influence new Ada programmers by presenting an approved set of interfaces and a programming style they can emulate.

Many functions can be included in package sets. The first is an integrated set of message generation and handling packages. Several packages can be prepared to deal with the ARPANET and AUTODIN network interfaces, while other packages can provide standard message generation and formatting facilities for inter-user mail, free-format (narrative), and formatted military messages. Packages can be provided for database management, text processing, and sort/merge functions.

Packages for use in embedded computer systems should receive special emphasis. Signal processing and navigation algorithms can be standardized in package sets after approval by a national verification procedure. Packages for graphics could contain routines for line drawing, manipulation of sub-screens within the output screen, character manipulation, three-dimensional object manipulation, shading, scaling, etc.

Packages would follow ANSI, FIPS, or ISO standards. A central group will be responsible for the coordination of package development teams and the standardization of interfaces documentation, etc.

A significant improvement in programming productivity in the relevant usage areas could easily be the result of the widespread availability of standard, reliable, well-documented, and easy-to-use

packaged software sets. In addition, good programming practices established for the code in the sets (enforced by strict management controls) would provide standards for programmers. It is estimated that moderate cost savings would result.

#### Some Relevant Research and Products

Proposed standards already exist for certain usage areas such as data base management and graphics. Attempts to find other candidates for common software packages are already underway. One example is by R. T. Chien and L. J. Peterson, Optimized Computer Systems for Avionics Applications, which describes significant commonalities among the kernels and algorithms used in radar, communications, and image processing.

#### Remarks on Rationale

One of the primary benefits of using Ada will be the exploitation and the fostering of commonality among various embedded computer systems; thus the emphasis on rigid definition of the language, portable compilers, and special language constructs for packaged software.

#### References

Chien, R. T., and L. J. Peterson. Optimized Computer Systems for Avionics Applications. AF Wright Aeronautical Labs, Wright-Patterson AFB, Ohio, report AFAL-TR-79-1235, 11 Feb. 1980.

Ichbiah, J. D., et. al. "Rationale for the Design of the Ada Programming Language." Sigplan Notices vol. 14, no. 6 (June 1979), Part B.

#### A.1.1.3 System Dictionary/Directory.

##### Description

Commercial data dictionary/directories have grown to include information about the entire system as well as data. This capability can be extended to include the information required for a complete and rigorous description of the system. The system dictionary/directory will become the organizer of all official knowledge concerning the system. It could also provide a system for maintaining unofficial notes and comments for limited periods. As the "memory" of an integrated software support environment, it will provide a framework for completely captured software. The Ada programming support environment data base capability can be used as a basis for implementation. The thrust will examine the potentials for the system dictionary/directory, determining the best ways to organize, manipulate, and use such a database.

The thrust must investigate methods for extending the database to include languages other than Ada and code in existing systems. The objective is to design a comprehensive, understandable system dictionary/directory that will include all needed information about a software system, regardless of the languages used or the system's history. Manipulation of the database must be a powerful and easily used feature, and specialized interfaces should provide for automatic data input to the database from tools such as language processors, linkers, and librarians, within the integrated software support system. Capabilities should be provided to ease the transition to the new database from previous automated or manual systems.

Because of the importance of a system dictionary/directory, work on specialized versions will certainly take place. DoD must ensure

that dictionary/directory systems be able to accept input from languages other than Ada and from systems that are already partially or completely implemented; otherwise, a great deal of the value of having a central dictionary/directory would be lost. If the dictionary/directory does not hold all system information, non-standard, isolated data files will again be built to fulfill the need. With all system information in an easy-to-use database, however, a strong positive impact on programmer productivity will result.

It is estimated that low-to-moderate cost savings would result from the implementation of the system dictionary/directory alone, without the additional tools in the integrated software support environment.

#### Some Relevant Research and Products

Limited system dictionary/directories, such as Datamanager and Data Catalogue 2, are already available, called "data dictionary/directories."

#### References

- Buxton, J. N., and V. Stenning. Requirements for Ada Programming Support Environments ("Stoneman"). Arlington, VA: DARPA, Feb. 1980.
- Curtice, R. M., and E. M. Dieckman. "A Survey of Data Dictionaries." Datamation vol 27, no. 3 (Mar. 1981), pp. 135-158.
- Pedersen, J. T., and J. K. Buckle. "Kongsberg's Road to an Industrial Software Methodology." IEEE Trans. Software Eng. vol. SE-4, no. 4 (July 1978), pp. 263-269.
- Rochkind, M. J. "The Source Code Control System." IEEE Trans.

Software Eng. vol. SE-1, no. 4 (Dec. 1975), pp. 364-370.

Wasserman, A. I. "Guest Editor's Introduction: Automated Development Environments." Computer vol. 14, no. 4 (April 1981), pp. 7-10.

#### A.1.1.4 Set(s) of Tools Covering Entire Lifecycle.

##### Description

In addition to ADA programming support environment tools (compiler, debugger, linker loader, editor, run controller, and configuration manager), sets of tools with carefully defined interfaces are needed to cover important features of each part of the lifecycle. Included in such sets might be simulators to simplify feasibility analyses, requirements languages, software specification languages, design languages, static analyzers for all languages used, formal verification tools, testing tools, change impact analyzers, and optimizers. Management aids for planning and control aids must also be included.

Developing more than one set of tools is a possibility, with each set based on a different methodological approach to software construction. Because this would conflict with the desire for standardization, it should only be done if methodologies are found with important, but incompatible, advantages.

The ISSE, the APSE, and the tool sets available within them should provide for: (1) a high degree of meaningful visibility into the system being supported, (2) a means for effective communication and coordination among all concerned, (3) easy collection and dissemination of information about the system, (4) efficient use of human resources, (5) assistance in improving system quality, (6) an appropriate diversity of views of a system for different purposes and audiences (for example, among the views might be control flow and dataflow), (7) a model of the system's performance or economic characteristics, (8) abstract or incomplete description and incremental development, (9) prototyping, (10) easy backtracking and modifi-

cation, (11) natural technical and managerial review points or milestones, (12) multiple versions, (13) usage history and statistics, (14) support of all aspects of software including documentation and test data, (15) facilities for expanding or enhancing the tool set, (16) high reliability, (17) good performance, (18) an easy in-migration path, and (19) an acceptable cost of operation.

A tool set should reflect, aid in learning, and encourage the use of its underlying methodological approach to software. It should conform to ASPE and ISSE standards and conventions, encourage good practices, and prohibit bad practices.

The tool set should be easy to learn; built-in and self-teaching training and documentation should be included. While a good production-quality tool set will be oriented toward experienced practitioners, attention should also be given to the role and training needs of novices.

Many individual tools have thrust candidates of their own. Nevertheless, an effort to ensure development of compatible tools would result in additional moderate cost savings throughout the life-cycle, in addition to the obvious benefit of reduced errors and greater continuity from phase to phase.

#### Some Relevant Research or Products

The tools in the Unix Programmer's Workbench (PWB) have a crude compatibility deriving from the byte-string nature of all Unix files. Any tool can read the output of any other tool, but meaningful processing is a different matter. Maestro, another tool set, has clear compatibilities and incompatibilities among its parts; however, neither Unix PWB nor Maestro contains the full spectrum of tools one would like. Some other tools sets of interest are CADES (ICL), USE (UCSF), Gandalf (CMU), and DREAM (Univ. of Colorado at Boulder).

Formal verification tools and methodologies for systems development that may or may not currently have software tools are also of interest. This is an area of active research and development in universities and in industry.

#### References

Buxton, J. N. "An Informal Bibliography on Programming Support Environments." ACM Sigplan Notices vol. 5, no. 12 (Dec. 1980), pp. 17-30.

Buxton, J. N., and V. Stenning. Requirements for Ada Programming Support Environments ("Stoneman"), Arlington, VA: DARPA, Feb. 1980.

Graham, R. M., chairman. "Proceedings of the ACM-SIGPLAN Symposium on the Ada Programming Language." ACM Sigplan Notices vol. 15, no. 11 (Nov. 1980).

Osterwald, L. J. A Software Lifecycle Methodology and Tool Support. U. Of Colorado at Boulder, Boulder, CO, report CU-CS-154-79, Apr. 1979. (NTIS accession no. AD-A076 335.)

Reifer, D. J., and S. Trattner. "A Glossary of Software Tools and Techniques." Computer vol. 10, no. 7 (July 1977), pp. 52-60.

Riddle, W. E., and R. E. Fairley. Software Development Tools. Berlin: Springer-Verlag, 1980. (See especially Chapter 1 and I. Nassi, "A Critical Look at the Processes of Tool Development: An Industrial Perspective.")

Wasserman, A. I., ed. "Special Issue on Automated Development Environments." Computer vol. 14, no. 4 (Apr. 1981), pp. 7-54.

#### A.1.1.5 Software Engineer's Support System.

##### Description

The Software Engineer's Support System would provide assistance and advice to software technicians and managers in the forms of intelligent reminders, prompts, elaborations, implication derivation, verification, warnings, and technical suggestions. For example, it could point out alternatives for implementation or suggest a similar previous effort from its records of prior projects. The system could function as an administrative aide, extending the user's ability to trace ideas, or it could perform common tasks with little direction. It could replace some functions presently performed by programmer aides, managers, quality assurance personnel, technical writers, as well as by the key software technical staff. It would embody what is currently contained in manuals and other reference materials, guidelines or standards.

Different application domains and different solution/approaches will require different knowledge, and different functions will require different procedures. Because the scope of such a capability is unclear, its development will occur through a number of interactions of construction and experience.

The Software Engineer's Support System will function in part as a tool set manager, deciding when to invoke tools and interpreting the results of their use. It will provide a way to propagate the best practices, allowing less skilled individuals to learn and produce better products. It would also provide continuity through personnel changes by providing a semi-intelligent interface with completely captured software.

The Software Engineer's Expert System offers significant improvement potential in the cost and speed of technology transfer of new skills and knowledge, which would otherwise be difficult to learn, and the quality of products produced by less skillful personnel. Other improvements might occur in productivity, in thoroughness, and in the ability of individuals to work alone on some tasks. The advantages of a "second opinion," particularly when personnel are not experts in an area, might be similar to those claimed for medical diagnostic software.

Since such a system will take a long time to develop, and since its detailed characteristics are uncertain, the benefits are difficult to assess; however, moderate cost savings can be expected in the development and operation phases of the lifecycle.

#### Some Relevant Research or Products

The Programmer's Apprentice and Spadee, both at MIT, are beginnings in this area. Barstow's knowledge-based programming effort is also relevant. Much Artificial Intelligence (AI) research is also relevant to learning, reasoning, intelligent support systems, and representation of knowledge.

The methodologies underlying set(s) of compatible tools covering the entire lifecycle will provide motivation for some of the support system's behavior.

#### Remarks on Rationale

Large software and computer-based systems are among the most complex artifacts ever attempted. Computer-based assistance in production and maintenance of software is a necessity. In the long run, automated intelligent assistance will be required to produce and

maintain a substantial number of large systems, because large numbers of talented personnel may not be available.

#### References

Barstow, D. R. Knowledge-Based Program Construction. Amsterdam: North Holland, 1979.

Miller, M. L. and I. P. Goldstein. "Planning and Debugging in Elementary Programming." In Artificial Intelligence: an MIT Perspective, P. H. Winston and R. H. Brown, eds., vol. 1, pp. 317-337. Cambridge, MA: MIT Press, 1979.

Rich, C., and H. E. Shrobe. "Design of a Programmer's Apprentice." Ibid. vol. 1, pp. 137-173.

Rich, C., and H. E. Shrobe. "Initial Report on a Lisp Programmer's Apprentice." IEEE Trans. Software Eng. vol. SE-4, no. 6 (Nov. 1978), pp. 456-467.

Rich, C., H. E. Shrobe, et. al. Programming Viewed as an Engineering Activity. Artificial Intelligence Lab, MIT, Cambridge, MA, memo AI-M-459, Jan. 1978. (NTIS accession no. AD-A052 307.)

Wilcox, T. R., A. M. Davis, and M. H. Tindall. "The Design and Implementation of a Table-Driven, Interactive Diagnostic Programming System." Commun. ACM vol. 19, no. 11 (Nov. 1976), pp. 609-616.

#### A.1.1.6 Programmer Workstation.

##### Description

The emergence of inexpensive processing power and memory allows programmer workstations to be provided with significant local capabilities. Advances in graphics and displays and in the understanding of human factors point to many potential improvements in the programmer/computer interface. Multi-media, multi-screen stations with various input devices could provide a powerful interface. One example in this direction is the "Spice" Workstation proposed by Carnegie-Mellon University.

A programmer workstation provides an interface between the programmer or software engineer and the computer system. Workstations are interconnected with centralized computational and data base management facilities. Their modularity in both hardware and software allows them to be modified to suit programmers' needs and to keep pace with the latest technology. Standardized features and interfaces of the workstation would decrease the retraining time for programmers assigned to a new project.

Considerable research is needed in this area. The hardware configuration of the workstation needs to be investigated to find those configurations combining low cost and modularity with high programmer productivity and ease of use. Workstation software must be specified and developed. The software must lend itself to modification, portability among hardware configurations, and rapid installation of varied sets of software tools and communications systems.

The current high level of interest in local area networks is a contributing factor behind the interest in programmer workstations,

and it should be relatively simple to interest groups in human factors research and standardization.

The largest improvement would occur in the effectiveness of the man-machine interface. Other benefits would include enhanced response time, reliability, and flexibility, along with ease in supplying uniform software support. The net cost savings for the workstation alone should be low; however, the use of the workstation with the integrated software support environment should increase productivity significantly.

#### Some Relevant Research/Products

Many types of programmer workstations are in use, under development, or under consideration. Examples are: the "Spice" personal scientific computing environment proposed by Carnegie-Mellon University [Proposal for a Joint Effort in Personal Scientific Computing, August 23, 1979]; work at the Xerox Palo Alto Research Center's Computer Science Laboratory on programming environments [see L. P. Deutsch and E. A. Taft]; the paper Personal Development Systems for the Professional Programmer by S. Gutz, A. I. Wasserman, and M. J. Spier; and the Problem Solving Environment discussion in the description of Project Quanta, submitted to the National Science Foundation by Purdue University and the International Mathematical and Statistical Libraries, Inc. on November 14, 1980.

#### References

Computer Sciences Department. Project Quanta. Computer Sciences Dept., Purdue University, West Lafayette, IN. 14 Nov. 1980.

Deutsch, L. P., and E. A. Taft, eds. Requirements for an Experimental Programming Environment. Xerox Palo Alto Research Center, Palo

Alto, CA, report CSL-80-10, June 1980.

Gutz, S., A. I. Wasserman, and M. J. Spier. "Personal Development Systems for the Professional Programmer." Computer vol. 14, no. 4 (Apr. 1981), pp. 45-53.

Wirth, Niklaus. "Lilith: A Personal Computer for the Software Engineer." In Fifth International Conference on Software Engineering, pp. 2-15. Los Alamitos, CA: IEEE Computer Society, 1981.

#### A.1.1.7 Useful Measures of Software Quality.

##### Description

Software metrics are needed to provide quantitative measures for important characteristics of software. The set of metrics must not encourage loophole exploitation for the sake of optimizing the response of the measurement system.

Software metrics have a number of potential uses. Minimum standards could be set for acceptance of deliverables, and incentives could be placed in contracts based on software quality measures. Competing designs or products could be compared to select the best. Metrics may be useful for predicting maintenance costs, reliability, portability, performance, and utility, both within intended application and elsewhere. Metrics can direct automatic or semi-automatic improvements of software quality, and aid in the training and evaluation of personnel. Thus, metrics can be used to specify, judge, improve, and predict. Measurement should be automatically performed; however, consistent measurement by humans is acceptable, if it can be used effectively.

Measures of merit will derive from a composite of more elementary metrics. Flow of control, data flow, semantic integrity, performance, and style are usually the most important categories of metrics. Trade-offs in system cost and characteristics will indicate the relative importance of different metrics within a measure of merit for different systems. Composites will be needed to avoid the loophole effect. "Engineering" or empirical measures are expected within the time frame of the Initiative; subsequently, measures based on theory may be established.

This thrust contributes to many other thrust candidates, including those related to management (especially acquisition management), software tools, training and evaluation of personnel, software quality improvement, maintenance, and verification and validation. Metrics developed in other thrusts will contribute to this thrust as well.

The greatest benefit could result from the assurance that acceptable quality software is being acquired. Many of the benefits of this thrust will be indirect in the sense of deriving from application in other thrusts. The chance of research and development success is fairly high, and the technology transfer penetration percentage is good. The net expected cost savings are moderate.

#### Some Relevant Research and Products

Software metrics is an area of research and development increasingly active in both universities and industry. Cost, complexity, and reliability have been the three dependent variables of greatest interest. The characteristics measured and the types of modeling and experimentation performed have varied. Halstead's software science approach is currently the only general theory that attempts to explain a wide range of phenomena. Walston and Felix at IBM used statistical techniques to analyze large bodies of data concerning software projects (but not the code) and to determine the effects of various characteristics. RADC's collections of data are aimed at similar work, as well as at investigating and validating models. Code measurements have been used in combination with real project data or in controlled experiments to study the relative predictive power of such measures as Halstead's and McCabe's, and such simple measures as lines of code, e.g. Bill Curtis' work at General Electric.

### Remarks on Rationale

Measurement is the beginning of engineering and science. Many engineering fields were successful for many years on limited theory but effective measurement. Likewise, useful software metrics can precede the existence or acceptance of a theory explaining software complexity.

### References

Basili, V. R., ed. Models and Metrics for Software Management and Engineering. Los Alamitos, CA: IEEE Computer Society, 1980.

Bowen, J. B. "A Survey of Standards and Proposed Metrics for Software Quality Testing." Computer vol. 12, no. 8 (Aug. 1979), pp. 37-42.

Fitzsimmons, A., and T. Love. "A Review and Evaluation of Software Science." ACM Computing Surveys vol. 10, no. 1 (Mar. 1978), pp. 3-18.

Halstead, M. H., ed. "Special Collection on Software Sciences." IEEE Trans. Software Eng. vol. SE-5, no. 2 (Mar. 1979), pp. 74-128.

McCabe, T. J. "A Survey of Industrial Software Quality Assurance." PRC Corp., McLean, VA, tech. note PRC 819-3, Oct. 1978.

McCall, J. A., and M. T. Matsumoto. Software Quality. vol. 1, Metrics Enhancement, and vol. 2, Measurement Manual. Rome Air Development Center, Griffiss AFB, NY, tech. report TR-80-109, Apr. 1980.

Ruston, H., chairman. Workshop on Quantitative Software Models. IEEE, New York, NY, catalog no. TH0067-9, October, 1979.

#### A.1.1.8 Multiple Representations of Software.

##### Description

Different representations of software can facilitate different considerations or manipulations of software. Just as different mathematical notations provide convenient tools for different types of problems or aspects of the same problem, different representations of software could facilitate different aspects of construction and maintenance.

Pictures, sound, and animation offer new ways to perceive and new chances to ensure that subtle errors are found. Various media, including graphics, text, and voice as well as static and dynamic representations, are all possibilities. Representations can be generated automatically from one another, or two separately prepared representations can be compared for consistency.

Rigorous documentation becomes a possibility as operator's, user's, or maintainer's "manuals" are automatically generated with guaranteed consistency. Requirements, program code, and other representations will be involved.

Most of the savings from this thrust will occur in the development and operations phases of the lifecycle. The savings will be due to faster error detection and clearer understanding of the system on the part of maintenance personnel. Savings are expected to be moderate in the affected phases.

### Some Relevant Research/Products

Softech is working on "living documentation," which incorporates multiple representations and methods for maintaining compatibility among them. Intermetrics is interested in notations in a variety of representations, and Hughes has done work with graphics in the Design Analysis System (DAS). I.P. Goldstein has done some AI work on recognizing conflicts between representations.

Some commercial and academic tool sets incorporate multiple representations. The Maestro programmer support environment automatically produces Nassi-Schneiderman diagrams, while computer-aided design and manufacturing efforts provide examples of multiple representations and multiple manipulation modes.

R&D difficulties exist in generating prose and in automatically converting between representations.

### References

Riddle, W. E., and R. E. Fairley. Software Development Tools. Berlin: Springer-Verlag, 1980.

#### A.1.1.9 Earliest Possible Error Detection.

##### Description

The cost of repairing an error is reduced if it is detected soon after it is committed. The goal of this thrust is to supply theory about the points at which various types of errors can be committed and detected (or equivalently, their absence verified), and to construct practical methods for verification and detection.

An example in one limited area is a syntax-knowledgeable editor, which informs a programmer of syntax errors when he is entering or changing a program. Errors can be detected in all aspects of software including requirements, design, and documentation. Questions to be addressed are: Where is the earliest theoretical point at which verification could be done? How might it be done? How could it be automated?

Studies have shown that the cost of fixing errors detected late in the software lifecycle is large. Benefits accruing from success of this effort are correspondingly large and would appear in the development and operations phases of the lifecycle. Reduction in the number of residual errors should also occur; however, some effort would be needed for the detection.

##### Some Relevant Research/Products

A number of efforts have been undertaken to check statements of requirements and designs for certain kinds of errors. Most prominent is the University of Michigan's PSL/PSA, which has been extended or revised in a number of places, including TRW, Boeing, Hughes, and the Army's BMD Advanced Technology Center. Other efforts have been

undertaken at High Order Software and at Computer Sciences Corporation.

Research in program proofs and other aspects of verification and validation is relevant. This thrust aims for a systematic basis for understanding the ultimate limits of error detection, and a mechanism to approach those limits.

#### Remarks on Rationale

The longer an error goes undetected, the more expensive it is to correct. For example, a design error which would cost one unit to correct during design may cost 4 units to correct during unit programming/testing, 9 units during component testing, 15 units during systems testing, and 30 units during operations. Errors from prior stages make up the bulk of the roughly 50% of the development and maintenance costs caused by errors; the next best thing to never committing an error is finding it as soon as possible.

#### References

ACM. "Workshop on Formal Verification." ACM Software Engineering Notes vol. 5, no. 3 (July 1980), pp. 4-47.

Buda, A. O. et al., "Implementation of the ALPHA-6 Programming System, IEEE Transactions." Academy of Sciences USSR, March, 1975, cited in D. S. Alberts, "The Economics of Software Quality Assurance," in E. Miller, ed., Tutorial on Program Testing Techniques, COMPSAC 77. Long Beach, CA: IEEE Computer Society, 1977.

Deutsch, M. S. "Tutorial Series - 7: Software Project Verification and Validation." Computer vol. 14, no. 4 (Apr. 1981), pp. 54-70.

Miller, E., and W. E. Howden, eds. Tutorial: Software Testing and

Validation Techniques. Los Alamitos, CA: IEEE Computer Society,  
1978.

#### A.1.1.10 Configuration Independence.

##### Description

Configuration independence implies that an application will run on various hardware systems and with different operating systems. Changes to the implementation of the application in these different environments, if necessary, will be accomplished automatically.

Typical constraints on the portability of software are memory size, word size, operating system incompatibility, programming language incompatibility, and incompatible utility program libraries (including sorting routines and database managers, etc.). Software will be more portable if standard languages without dialects are used; if languages compel explicit specification of the range of integers and the precision of floating point values; if virtual memory is provided; if a standard operating system interface (at some level) is defined; and, if libraries of standard, portable, utilities are provided.

For each different operating environment, an automated mechanism should indicate whether a given application program is acceptable to it, and what special cost (e.g. multiple precision arithmetic) would be incurred by running the application in this environment. Alternatively, for each operating environment, a mapping to a standard, "virtual" operating environment and a mapping from this standard environment could be devised. Other possibilities for achieving configuration independence are language processor features that recognize the need for multiple precision arithmetic and generate the appropriate function calls where needed; standard interfaces between applications and hardware, especially for interactive or real-time devices; and a system design regimen in which configuration-dependent

software is isolated into application-independent modules (therefore, replaceable by existing equivalent modules in the new configuration).

To establish configuration independence, the boundaries of hardware and systems software must be defined, as well as methods for dealing with changes. Layered designs to provide stable layer interfaces and application generators to tailor the application to each configuration are both approaches to the problem.

The principal benefit would be the ease of transferability of software between systems and the consequent savings in software development costs. Configuration-independent software could be implemented on a variety of modular systems. As a result, hardware could be chosen to fit a particular system's projected workload, rather than to accommodate some equipment-dependent or configuration-dependent software. It would also be easier to take advantage of improved hardware designs. The real savings come from not having to redevelop/convert to run on other configurations or in other environments.

Cost savings are expected to be moderate and to appear primarily in the development phase of the lifecycle.

#### Some Relevant Research/Products

The National Software Works is an experiment to construct libraries of transportable programs, and to standardize some operating system functions. The efforts of ANSI to define standards for COBOL, FORTRAN, character sets, etc. have been of great assistance. The ADA effort, the ADA support environment, and the Military Computer Family (MCF) common architecture projects are evidence of what can be achieved in the interest of portability.

### Remarks on Rationale

Software is so expensive and the required personnel resources so scarce that to spend large amounts of effort and funds to change from outdated environments/hardware to new ones is a waste when achievement of a measure of configuration independence is possible. On the other hand, the hardware and environment must provide for change in some way, if new technology is to be exploited.

### References

Collica, J., M. Skall, and G. Blotsky. Conversion of Federal ADP Systems: A Tutorial. National Bureau of Standards, Washington, DC, report NBS 500-62, Aug. 1980.

Ebert, R., J. Lugger, and R. Goecke, eds. Practice in Software Adaptation and Maintenance. Amsterdam: North-Holland, 1979.

### A.1.2 Conception/Feasibility

#### A.1.2.1 Rapid Simulation.

##### Description

Alberts reports that more than five years often pass between a DoD system's conception and the start of its formal requirements definition. Capabilities are needed for the rapid simulation of newly conceived systems to judge their probable throughput, error rate, costs, etc. Rapid simulation will assist in early conceptual understanding, and speed decisions concerning implementation.

Rapid simulation of a proposed new system or of a proposed change to an existing system will speed decision-making while reducing risk. Relevant performance and cost-effectiveness measures will be obtainable. Expected values and measures of risk and sensitivity usually must be supplied. Both hardware and software considerations are relevant; however, because the simulation of software is less well understood, R&D efforts will emphasize software issues.

The specific requirements of a rapid simulation capability must be more clearly identified and understood. Integration of developed tools into DoD practices must also be addressed. Tools must be developed and validated to give decision-makers confidence in them. The tools may vary with the problem domain and the basic technologies utilized.

The most significant improvement potential exists in the time required for analysis and decision-making during the conceptual phase. Other benefits may occur during the design and operation periods. Significant cost savings are expected from earlier and more

complete predictability of results. In addition, the time for the conceptual phase might be reduced, resulting in considerable benefits to users.

Benefits will also accrue to organizations outside of DoD that wish to simulate similar systems. How widely useful the tools developed will be depends on how customized they are to DoD-unique situations.

#### Some Relevant Research and Products

Substantial activity is being performed under the title of computer performance evaluation (CPE). Most of the emphasis, however, has been on hardware and systems software performance. CPE work is just beginning on applications software. Serious developments in cost estimation methodologies are also in their infancy, but some software packages are in existence and promising research is underway.

DoD information systems and software are often large, complex, and difficult to evaluate, especially before construction. This uncertainty leads to long analytical and decision-making activities that delay the system. Simulation of proposed concepts is within reach to speed up these activities. Rapid simulation will be useful for comparing alternatives during design, and for evaluating proposed changes during operation.

#### References

Alberts, D. S. "The Economics of Software Quality Assurance." in 1976 National Computer Conference, AFIPS Conference Proceedings, vol. 45, pp. 433-442. Montvale, NJ: AFIPS Press, 1976.

Computerworld. "'Crystal' Clarifies, Predicts Software Performance Problems." Computerworld vol. 14, no. 52 (December 22, 1980), p. 43.

Myers, W. "Conference Report: 1978 Summer Computer Simulation Conference." Computer vol. 11, no. 10 (Oct. 1978), pp. 70-74.

Schneidewind, N. F. "The Use of Simulation in the Evaluation of Software." Computer vol. 10, no. 4 (Apr. 1977), pp. 47-53.

### A.1.3 Requirements

#### A.1.3.1 Rapid Prototyping.

##### Description

Since requirements are often difficult to formulate when similar automated applications do not exist, a quickly constructed prototype of a new application can be used to derive the full system requirements. Rapid prototyping will alleviate the problems users and analysts encounter in trying to specify fully the requirements for a system. Rapid prototyping can both reduce the time needed to produce requirements, and improve the quality of the requirements eventually produced.

Rapid development of quality system and software requirements is possible with the use of prototype systems. To provide a useful rapid prototyping ability for DoD applications, processing functions, data collected and stored, and user interfaces need to be identified. These may vary among applications and with user tolerance and imagination.

Questions to answer are: What can best be entirely ignored in prototypes or handled without concern for details? For what requirements issues is prototyping a suitable aid in defining answers? In what language or other medium should the prototype system be represented? How should the prototyping process proceed? When and how should prototyping be terminated and a production quality system produced?

This capability is related to very high level languages and requirements languages. It shares description requirements with

rapid simulation tools. It differs from rapid simulation in that its output is a functional subset of a system; rapid simulation produces numbers from which feasibility can be determined.

The greatest potential exists for quality of requirements. The principal increase in quality will be in the improved relevance and usefulness of the system, as well as reduced errors in requirements. Other benefits may include quicker development of requirements and better mutual understanding between users and developers.

The level of technology transfer penetration is dependent on software acquisition and development practices. There may be no net cost savings in the development stages, but system utility should be greater during the operation stages.

#### Some Relevant Research and Products

General data base management systems with simple but powerful interfaces are making claims in this area already; e.g. NOMAD. Of the responses to the 30 July 1980 announcement in the CBD, seventeen claimed competence in the area of rapid prototyping, and eight had actual projects or products.

#### Remarks on Rationale

Rapid prototyping is promoted as a solution to a major problem: users of a system who do not have experience with similar systems find it difficult to conceive what they will need; therefore, they cannot evaluate alternatives accurately. By providing similar, relevant experience, prototypes help users think about systems requirements.

Learning from prototypes is an established technique in hardware engineering. In electrical engineering, breadboard and brassboard

prototypes are common. Software systems should be developed analogously.

#### References

Dodd, W. P. "Open Channel: Prototype Programs." Computer vol. 13, no. 2 (Feb. 1980), p. 81.

Gomaa, H. and D. B. H. Scott. "Prototyping as a Tool in the Specification of User Requirements." In Fifth International Conference on Software Engineering, pp. 333-339. Los Alamitos, CA: IEEE Computer Society, 1981.

McCracken, D. D. "Software in the 80's: Perils and Promises." Computerworld vol. 14, no. 38 (Sept. 17, 1980), pp. 5-10.

Zave, P. and R. T. Yeh. "Executable Requirements for Embedded Systems." in Fifth International Conference on Software Engineering, pp. 295-304. Los Alamitos, CA: IEEE Computer Society, 1981.

#### A.1.3.2 Application Domain Expertise.

##### Description

Excellent systems are the result of computing expertise coupled with application area expertise. Identification of commonalities in DoD applications can lead to reusable software. For example, inertial navigation might be found to be a common function across a wide variety of platforms and systems, or radar and other electromagnetic sensors may have similar target information extraction functions. For each commonality, economies are possible from software generators, standard software components, standard designs, or capture of application knowledge that can be re-used.

The complete taxonomy of DoD software functions will allow better judgments concerning proposed standards. The relative importance of different types of activities and software functions will help guide R&D investment.

Application knowledge is one form of expertise that will be needed in the knowledge-based systems planned for the long term. This is true (although varying somewhat) in software tools for construction and maintenance, and in application systems themselves. Other efforts that could benefit include user-oriented requirements languages, standard Ada package sets, and high-confidence testing.

This thrust would aid recognition and exploitation of commonalities and the task of setting the overall direction of software technology R&D. It could also forestall duplicate investigation and analysis across the many thrusts requiring application domain expertise.

Most of the cost saving benefits of this thrust will be received indirectly through other thrusts; therefore, its direct cost saving

benefits are estimated to be low. Most of the benefits will occur in the development and maintenance phases of the lifecycle.

#### Some Relevant Research and Products

The Ada requirements development demonstrated a substantial amount of commonality among embedded systems at the programming language level, although this does not imply commonalities at the functional level. Function lists have been developed for C3 systems, and efforts such as COMTEC-2000 have tried to factor in application area needs. Avionic commonalities have been studied by Chien and Peterson.

#### References

Barstow, D. R. Knowledge-Based Program Construction. Amsterdam: North Holland, 1979.

Belady, L. A. "Evolved Software for the 80's." Computer vol. 12, no. 2 (Feb. 1979), pp. 79-82.

Carlson, W. E. (Chairman). Defense Computer Resources Technology Plan. Washington, DC: R&D Technology Panel, Management Steering Committee for Embedded Computer Resources, USD(R&E), June, 1979.

Chien, R. T., and L. J. Peterson. Optimized Computer Systems for Avionics Applications. AF Wright Aeronautical Labs, Wright-Patterson AFB, Ohio, report AFAL-TR-79-1235, 11 Feb. 1980.

COMTEC-2000 Study Group. Computer Technology Forecast and Weapon Systems Impact Study (COMTEC-2000). 3 vols. HQ Air Force Systems Command, Washington, DC, tech. report 78-03, Dec. 1978 - July 1979.

#### A.1.3.3 Data Validation.

##### Description

The validity of data stored and reported by computer-based systems is a key factor in their acceptance by and utility to users. Currently, the editing or validation of input and stored data is without an underlying theory, and is elementary in practice, consisting mainly of type checking, range checking, simple comparisons, and arithmetic relationships. Data validation could benefit, if viewed as a problem in which existing automated decision-making methodologies could be applied. The intelligent application of these techniques offers more powerful abilities to deal with complexities and uncertainties.

The objective of this effort is to develop and demonstrate significantly better practical, automated techniques for editing input data and for validating stored data. Automated decision-making techniques, including Artificial Intelligence, will be explored and prototyped. Techniques and methodology for applying them in practice will be addressed.

The primary benefits will be in improved data accuracy and increased user confidence. Cost savings could be substantial, but not in what would normally be thought of as software expenses. Rather, savings would come in clerical and user time. The cost savings in traditional software personnel expenses will come from the rationalization of the input editing process and possibly from resulting reusable software. Benefits will occur in the development and operations phases of the lifecycle.

Data quality is an important determinant of overall system quality. Because of the lack of research, particularly by academia, the potential for fundamental improvement is high.

#### Some Relevant Research and Products

What little research that has been done in this area has been performed in the last four years, with most published in the last year. It has concentrated on simple descriptions of the problem, the possibility of using first order predicate calculus to express the traditional edits, and the potential for avoiding customized programming. None of these methods offers fundamental improvement in the quality of data stored in computer-based systems.

An active DoD leader in the area is the COPE effort at the Naval Research Laboratory.

#### Remarks on Rationale

Poor data is a widespread problem from which no computerized system is immune. The effectiveness of automated systems, and user acceptance of them, is open to significant improvement. In addition, rigorous research will be encouraged in an area with deep problems, but high payoffs.

#### References

Bernstein, P. A, B. T. Blaustein, and E. M. Clarke. "Fast Maintenance of Semantic Integrity Assertions Using Redundant Aggregate Data." In Proceedings, Sixth International Conference on Very Large Data Bases, pp. 126-136. Los Alamitos, CA: IEEE Computer Society, 1980.

Lee, R. C. T., J. R. Slagle, and C. T. Mong. "Towards Automatic Auditing of Records." IEEE Trans. Software Eng. vol. SE-4, no. 5 (Sept. 1978), pp. 441-448.

Necolas, J. M., and K. Yazdanian. "Integrity Checking in Deductive Data Bases." in Logic and Data Bases, H. Galliare and J. Minker, eds., pp. 325-344. New York: Plenum Press, 1978.

Taylor, D. J., D. E. Morgan, and J. P. Black. "Redundancy in Data Structures: Improving Software Fault Tolerance." IEEE Trans. Software Eng. vol. SE-6, no. 6 (Nov. 1980), pp. 585-594.

Wilson, G. A. "A Conceptual Model for Semantic Integrity Checking." In Proceedings, Sixth International Conference on Very Large Data Bases, pp. 111-125. Los Alamitos, CA: IEEE Computer Society, 1980.

#### A.1.3.4 Built-In Testing.

##### Description

Built-in testing capabilities can be based on suites of standard test data and test results, runtime checking of assertions, or comparisons of results for different redundant processes. This thrust will create software capabilities analogous to built-in testing capabilities in electronic hardware.

As computer systems and software packages grow larger, it becomes more difficult to guarantee their correct operation. Built-in testing is one way of ensuring that incorrect operation will be detected. Tests will detect not only errors in the original design but also those added during program changes.

Several test methods are currently under examination: assertions, dual program comparison, and standard test suites. Assertions are statements inserted into code to cause the state of designated variables to be tested during execution to ensure that values are within expected ranges. Assertions can state the expected relationship among variables as well as their individual values; for example, an assertion could state that the items in a list must always be in alphabetical order. Through the use of assertions, the programmer can ensure that unusual sequences of operations or incorrect code changes will not result in undetected failure of an important relationship. The use of assertions is still new, however, and it is difficult to determine assertion statements' optimum placement and content.

Dual program comparison also can be used to detect errors in the original code and later modifications. Two groups of programmers each write the code for a function, working from the same specifica-

tions. The two resulting modules are executed in parallel, and their outputs compared. Any discrepancies in the outputs are flagged as errors. Because the code in the two parallel modules will probably be different, the probability of detecting an error resulting from an unusual combination of inputs will rise. Costs appear to be higher for this approach; it is necessary to determine if the greater probability of error detection is worth the higher development costs and the extra complexity caused by the need to synchronize and compare module outputs. Higher costs may be offset by increased reliability and fewer system failures in the operational stage.

Standard test suites are sets of test inputs and expected outputs that could be run as background tasks during normal system operation, at start-up, or when the "test" button is pushed. Tests are selected to check all important hardware and software functions, and are used primarily to detect failed hardware or erroneous software "corrections". One use for test suites is the detection of surreptitious changes to software and hardware.

Various methods of built-in testing will be investigated, and their relative advantages and cost/benefit ratios evaluated. The primary benefit lies in the area of increased reliability. Until the testing discipline is more fully developed, the methods developed by this thrust may be crucial in the effort to produce more reliable software. This thrust will also assist in the on-line, continuous reverification of computer security mechanisms.

#### Relevant Research and Products

The use of built-in testing is common in electronic hardware from the integrated circuit level up through the systems level. Use of assertions has been widely advocated but less widely attempted for real-world embedded software. The use of permanently maintained test data suites is also fairly common, but arrangements for "push button"

retesting are unusual. Redundant software has been used for some critical applications such as nuclear reactors and the space shuttle.

#### References

Geiger, W., L. Gmeiner, H. Trauboth, and U. Voges. "Program Testing Techniques for Nuclear Reactor Protection Systems." Computer vol. 12, no. 8 (August 1979), pp. 10-18.

Miller, E., and W. E. Howden. Tutorial: Software Testing and Validation Techniques. Los Alamitos, CA: IEEE Computer Society, 1978.

Panzl, D. J. "Automatic Software Test Drivers." Computer vol. 11, no. 4 (April 1978), pp. 44-50.

#### A.1.3.5 Forgiving Systems.

##### Description

"Robust", "fault-tolerant", "user-friendly", "fail-soft", "high integrity", and "humanized" describe systems that are attractive and safe to use, despite mistake-prone human users, ubiquitous residual software bugs, inevitable hardware failures, and adverse environmental events. Although this candidate area will pursue solutions for all four types of problems, the emphasis will be on those caused by user errors. The goal is software systems that avoid disaster and help users obtain their goals despite any miscues.

Other types of systems, particularly human organizations, will be investigated to learn how they achieve their robustness and tolerance for error. Theory will be sought to aid in designing systems. The repertoire of techniques and approaches to provide forgiveness would be expanded and systematized. Among the issues of interest will be quantitative probabilistic judgments of reasonableness, predictions and quantification of error consequences, redundant data and software, back-up and recovery, algorithmic robustness, and human factors.

This candidate is related to the high data integrity and built-in testing candidates, and other candidates related to reliability.

##### Some Relevant Research and Products

A number of time sharing systems contain defenses against common user-caused disasters: for example, double checking that the user really wishes to have changes to a file ignored, or to have all his files deleted.

Fault tolerance for hardware failures has received considerable attention, as has system survivability. Redundant software has been used in such applications as nuclear reactors. In an interesting effort by the Jet Propulsion Laboratory (JPL) for a Voyager mission, ground commands were checked by the spacecraft to ensure that potentially damaging maneuvers were not performed. (An override ability was available to the ground and was used at least once.)

Natural languages and human organizations have been investigated from a number of perspectives, some of which are relevant to the concerns here; for example, the importance of assumptions, context, and pragmatics in the handling of exceptional conditions. New analyses and interpretations are needed.

The principal benefits will be increased user acceptance and confidence in systems and a reduction in major operational disasters. The development of systems may be more difficult, but their operational utility will be enhanced. The results achieved can be applied to software environments and tools, as to well as application systems. Benefits in predictability and reliability will follow.

The applicability of the results should be wide. Reduced costs for software development are not expected, but maintenance efforts may be reduced. System reliability and utility should be enhanced.

#### Remarks on Rationale

Blind obedience may have its value, but can easily lead to disaster in human organizations. Yet blind (or at least myopic) obedience is a principal characteristic of most software systems. If persons with this characteristic would be unacceptable for most important tasks, why should a computerized system with it be any more acceptable?

It will probably be a long time before systems can be built that have the speed and accuracy associated with computers and the ability to save us from ourselves associated with intelligent and considerate co-workers. Nevertheless, much improvement can be made over prevailing practices.

#### References

Glib. T., and G. Weinberg. Humanized Input: Techniques for Reliable Keyed Input. Cambridge, MA: Winthrop, 1976.

Hayes, P., E. Ball, and R. Reddy. "Breaking the Man-Machine Communications Barrier." Computer vol. 14, no. 3 (Mar. 1981), pp. 19-30.

Hecht, H. "Fault-Tolerant Software for Real-Time Applications." ACM Computing Surveys vol. 8, no. 4 (Dec. 1976), pp. 391-407.

Ledin, G. Jr., and V. Ledin. The Programmer's Book of Rules. Belmont, CA: Wadsworth, 1979, ch. 1, pp. 2-17.

Morgan, D. E., and D. J. Taylor. "Special Feature: A Survey of Methods for Achieving Reliable Software." Computer vol. 10, no. 2 (Feb. 1977), pp. 44-53.

Rasmussen, J. "The Human as a System Component." In Human Interaction with Computers, H. T. Smith and T. R. G. Green, eds., pp. 67-96. London: Academic Press, 1980.

Shneiderman, B. "Human Factors Experiments in Designing Interactive Systems." Computer vol. 12, no. 12 (Dec. 1979), pp. 9-19.

AD-A102 180

MITRE CORP MCLEAN VA

F/G 5/1

CANDIDATE R&D THRUSTS FOR THE SOFTWARE TECHNOLOGY INITIATIVE.(U)

MAY 81 S T REDWINE, E D SIEGEL, G R BERGLASS F19628-81-C-0001

UNCLASSIFIED

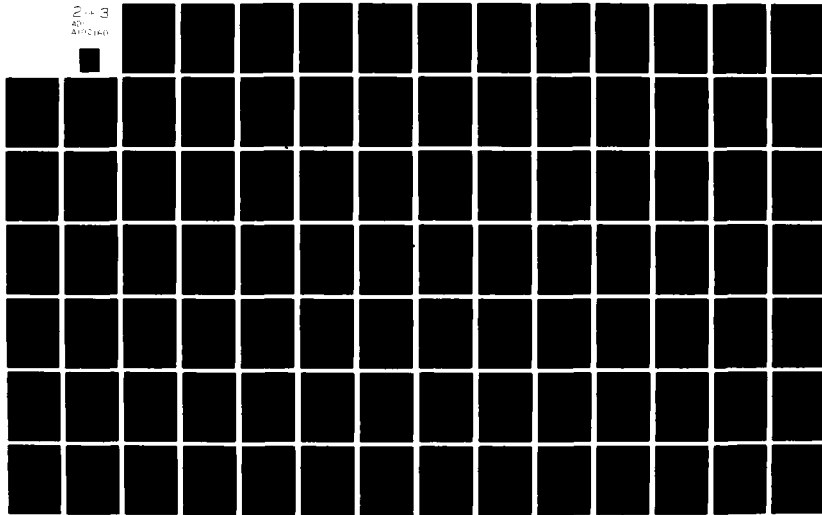
MTR-81W00160

NL

2 of 3

AD

A102180



#### A.1.3.6 User-Oriented Requirements Interfaces.

##### Description

Requirements languages describe what the software system must do, completely and unambiguously. To be used readily and to provide confidence that the statement of requirements is correct, the language must allow a natural statement of the requirements in terms that specifiers and requirements certifiers can understand and establish as correct. The need is for descriptions that are both natural and rigorous.

Variation among different application domains within DoD will necessitate a number of different requirements languages or dialects to provide natural requirements descriptions for different domains. Application expertise will be required to design requirements languages for each problem domain.

Requirements languages typically are not executable. Compilers constructed to make them executable with acceptable efficiency are often called Very High Level Languages (VHLL's). Since requirements languages are not acceptably executable, an executable solution written in a programming language is prepared. The executable solution must be verified for agreement with the requirements language description. Formal techniques must be used, if rigorous verification is required.

An interesting and unusual approach to user-oriented descriptions involves construction of an application model to which information system functions could be related. Some functions might be passive (e.g. observing and reporting on inventory levels); some might be replacements in well understood roles (e.g. replacing postal mail by telecommunications); some might address the less understood roles

(e.g. intelligence fusion and prediction); and some might be new functions. This "model of the world" approach is often used informally as an aid to communication and problem solving by user and developers.

Major improvements in the accuracy of communication between conceivers and developers, with corresponding reductions in the number of errors in requirements and the number of deviations from requirements in the final software are expected from the success of this effort. Only the first of these benefits will derive solely from the requirements languages; the other two will require additional tools (requirements language analyzers and formal verification aids) to reach their full potentials. Benefits could also accrue during testing and operation. Testing would benefit considerably from having formal requirements from which to generate test data. In the operations phase, maintenance of the requirements along with the rest of the software would create a baseline for verification of system changes.

Moderate cost benefits would occur in the development and operations phases of the lifecycle. R&D success is uncertain, because a number of different languages and dialects may be needed. The technology transfer penetration percentage will be only fair, because even good formalisms are avoided by many persons. Despite the low probability of R&D success and the fair technology transfer percentage, this thrust is still important, since it is a prerequisite for many other thrusts, including Formal Verification and other attempts to produce high reliability.

### Some Relevant Research and Products

Various problem-oriented languages have been developed, for example, COGO in civil engineering. Specification languages such as SPECIAL, GYPSY, and INA JO exist in connection with formal verification efforts. The specification language ANNA is attracting support as a partner to Ada. W. Martin at MIT has built a question and answer system which defines requirements (and generates systems) within the domain of inventory systems. Computer aided design (CAD) systems have some analogies to this thrust and often use graphic interfaces/notations. VHLL's have great relevance. A recent effort to respecify the A-7 avionics software is a significant research effort relevant to this thrust.

### Remarks on Rationale

Describing the problem/requirements in a natural but rigorous fashion understandable to users who can certify its correctness would provide a firm foundation for system development.

### References

Arden, B. W., ed. The Computer Science and Engineering Research Study (COSERS). Cambridge, MA: MIT Press, 1980, "Special Purpose Languages," pp. 589-602.

Heninger, K. L. "Specifying Software Requirements for Complex Systems: New Techniques and Their Applications." IEEE Trans. Software Eng. vol. SE-6, no. 1 (Jan. 1980), pp. 2-13.

Ross, D. T., ed. "Special Issue on Requirements Analysis." IEEE Trans. Software Eng. vol. SE-3, no. 1 (Jan. 1977).

Thurber, K. J. Tutorial: Computer System Requirements. Los Alami-  
tos, CA: IEEE Computer Society, 1980.

#### A.1.3.7 Complex Knowledge-Based Systems.

##### Description

Knowledge-based systems use highly organized knowledge bases (linked data structures) and flexibly incorporated inexact logic (heuristic rules) to deal with complex interrelationships. These techniques allow systems to solve complex problems through deduction, inference, and interaction with a user. Although knowledge-based systems are becoming larger, the most successful systems have dealt with well-defined, limited problem domains.

Knowledge-based systems could support all phases of DoD operations. The problems are to define appropriately limited domains, to organize knowledge about these domains, to construct the heuristics, and to find a hardware and software architecture on which these systems can run effectively. The ultimate goal is systems able to operate in various complex domains and across domains.

A system that can interact with a "user" (e.g. software engineer or logistics commander), make inferences and deductions from the context in which the user is working, and draw upon its "knowledge" of standards, conventions, rules, capabilities, and similar problems encountered previously has obvious utility. These systems are sometimes called "expert" systems, because they give the kind of assistance (at least in theory) that is expected from an experienced, knowledgeable assistant.

This candidate will explore means to provide automatic (artificial), intelligent support to DoD in various operational domains. How to "scale up" from current capabilities--whether, for example, some new machine architectures would help--is a major problem; also,

a substantial effort will be required to collect and organize knowledge in these domains.

The potential benefits are immense, as more human expertise is built into systems, and systems acquire the capability to make inferences and deductions in broader, more complex domains. There are no bounds on what can rationally be hoped for, although the mechanisms for achieving significant capability are largely unknown.

#### Current Research and Products

The first successful knowledge-based system was DENDRAL, developed at Stanford University. This program used heuristics developed by chemists to propose and verify the molecular structure of a class of organic compounds from mass spectroscopy and magnetic resonance data. The heuristics were used to select a few candidates from the millions of possible candidates. The system performed better and faster than the expert chemists who designed the heuristics; it even discovered errors and omissions in the literature.

MYCIN is a blood disease treatment program also developed at Stanford. Heuristic knowledge obtained from expert diagnosticians is used to suggest diagnoses, recommend further tests, and offer treatment alternatives.

DENDRAL and MYCIN were carefully constructed for limited, well-defined problem domains. This was necessary due to the absence of a general method for organizing the knowledge and rules of systems. The use of "frames" is a promising approach to this general problem, first suggested by Minsky. A frames system may be described as a set of interrelated templates, each describing part of some known type of situation. The templates describe the essential characteristics of a situation, but omit details that might be unique for a particular instance of that type of situation. An important point is that a

frame may contain defaults and procedural constraints for its empty details; thus, the data base itself can contain the heuristics that would be invoked automatically whenever data is entered or accessed. This type of organization, which can be used for both information and rules, provides a structure for handling larger, poorly defined problem areas.

#### Remarks on Rationale

Knowledge-based systems are seen as one of the keys to the solution of the complex problems of the future. The prospect of automated human thought processes, even automated inventiveness, with vast and organized memories seems farfetched, but the potential payoffs are enormous.

#### References

Birk, J., and R. Kelley. Research Needed to Advance the State of Knowledge in Robotics. In Proceedings of workshop at Newport, RI, 15-17 April, 1980. Washington, DC: National Science Foundation, 1981. (NTIS accession no. PB 81-132 557.)

Buchanan, B., G. Sutherland, and A. Feigenbaum. "Heuristic Dendral: A Program for Generating Explanatory Hypotheses in Organic Chemistry." In Machine Intelligence 4, Meltzer and Michic, eds. New York: Wiley, 1969.

McCarthy, J., T. Binford, D. Luckham, Z. Manna, and R. Weyhranch. Final Report: Basic Research in Artificial Intelligence and Foundations of Programming. Stanford A. I. Lab, Stanford, CA, Memo AIM-337, May 1980.

Minsky, M. "A Frame Work for Representing Knowledge." In The Psychology of Computer Vision, P. H. Winston, ed. New York: McGraw-Hill,

1975.

Pylyshyn, Z. W. "Artificial Intelligence." In What Can Be Automated, B. W. Arden, ed. Cambridge, MA: MIT Press, 1980.

Shortliffe, E.H. Computer-Based Medical Consultations: MYCIN. New York: American Elsevier, 1976.

Winston, P. H., and R. H. Brown, eds. Artificial Intelligence: An MIT Perspective. 2 vols. Cambridge, MA: MIT Press, 1979.

#### A.1.4 Design

##### A.1.4.1 Data Flow Approach.

###### Description

The Data Flow concept perceives a system in terms of data undergoing transformations, rather than procedures transforming data. The distinction is that data, not procedures, play the central role. This facilitates application description in an environment where there is no single control stream, e.g. a multi-computer system. A typical Data Flow graphic representation shows data flowing among bubbles signifying data transformations.

Multi-computer systems employing microprocessors hold promise of cost-effective solutions to time-critical problems, but it is hard to orient an application to a multi-computer environment. Because a Data Flow description is not unnecessarily sequential, it appears adaptable to the multi-computer environment.

A Data Flow description of an application is a useful tool for explaining the operation of a proposed system to its potential users, perhaps because it deals with data, with which the user is familiar, rather than the procedures of the systems analyst. Furthermore, a Data Flow description is another representation of an application, which may provide additional insight for its proper implementation. The relative value of Data Flow among the various approaches to software, including the functional and predicate approaches, needs investigation.

If it facilitates the adaptation of algorithms to multi-computer systems, Data Flow design methods may be a way to realize the

increased speed, lower cost, and smaller physical dimensions (for microprocessors) of these systems. If it helps designers to describe a proposed system concept, it may reduce the number of user-requested changes after the system becomes operational. If it helps systems analysts to understand the system, it may reduce the time and expense required to correct problems and effect improvements.

#### Some Relevant Research/Products

A data-oriented language, VAL, has been developed jointly by MIT and Lawrence Livermore Laboratory. Some scientific applications are being programmed in VAL to determine its usefulness, and there are similar projects elsewhere. There is no evidence that any kind of data flow system is currently in production use; however, the possibilities of multi-computer systems with many asynchronous processors are only beginning to be apparent. Data Flow design has been used as a descriptive and analytical tool in commercial software organizations for over five years.

#### References

DeMarco, T. Structured Analysis and System Specification. New York: Yourdon Press, 1978.

Denning, P. J. "Operating Systems Principles for Data Flow Networks." Computer vol. 11, no. 7 (July 1978), pp. 86-96.

Dennis, J. B. "Data Flow Supercomputers." Computer vol. 13, no. 11 (November 1980), pp. 48-56.

McGraw, J. "Data Flow Computing: Software Development." IEEE Trans. on Computers vol. C-29, no. 12 (December 1980), pp. 1095-1103.

#### A.1.4.2 Self-Interfacing Software.

##### Description

Software oriented towards functions that can be manipulated and combined with little attention to their interfaces is convenient for a designer. Some languages are currently oriented towards a particular data type and offer this capability for one type of data: e.g. arrays in APL and lists in LISP. For software in DoD application domains to acquire this capability, the ability to interface with a variety of data types is required. An automatic interfacing capability built either into the individual functions or into related tools (such as function builders) will allow the designer/programmer to ignore interfacing details without penalty and enhance his productivity.

Functions and components will be needed in suitable types and aggregates for application domains of interest. High level functions will provide the greatest increase in productivity. The first successful functional programming capability will probably be based on either a deep mathematical treatment of the function concept, or on a substantial expansion of "generic" function definition capabilities in present-day languages (e.g. Ada).

An alternative is a try/expose/fix programming method in which the original programming is tried in a functional form, interfacing difficulties beyond the machine's ability are reported, and the programmer supplies the required fixes. This is a variant on interactive component tailoring and interfacing.

Moderate cost savings in the development and operations phases of the lifecycle could result from this initiative. Software design

and change time would be markedly reduced, improving programmer productivity.

#### Some Relevant Research or Products

APL and LISP have relevant qualities, as do the UNIX operating system and the Consistent System developed at MIT and maintained by Renaissance Computing, Incorporated. Interest is increasing in functional programming languages, inspired in part by John Backus's 1977 Turing Lecture. A conference is scheduled on the subject for September 27-October 1, 1981, in Portsmouth, N.H., sponsored by ACM, ASIGPLAN, SIGARCH, SIGOPS and the MIT Laboratory for Computer Science. Research in denotational semantics may also be relevant.

#### Remarks on Rationale

The functional approach is one effort to provide a powerful conceptual approach and notation while diminishing details. It could revolutionize the way software and hardware are built.

#### References

Backus, J. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs." Commun. ACM vol. 21, no. 8 (August 1978), pp. 613-641.

Henderson, P. Functional Programming. Englewood Cliffs, NJ: Prentice-Hall, 1980.

#### A.1.4.3 Predicate Approach.

##### Description

The predicate (or axiomatic) approach has gained currency as a conceptual tool in software construction from a wide variety of sources. While alternative ways of thinking about programming are starting to receive attention, this approach still occupies a favored position in current programming research and development thinking.

The use of predicates (logical expressions of the status of data) to describe a program's state before and after a procedure's execution has been recommended for a decade. Many programming proof techniques and suggestions for computing program structures and styles have resulted from this approach. While formal proofs of programs remain realistic possibilities only for small programs, the impact of program proof efforts has been in the better understanding of specification, design, and informal verification techniques. Structured programming and the use of assertions are outgrowths of research in this area.

Tools and training aids need to be developed to aid the technology transfer of developed results. Much of what is needed to make the predicate approach relevant and to put it into wider use is based on existing research results, rather than new research. Most of the benefit will come from tool development and technology transfer.

Expanding the use of predicates to documentation and to guards for data bases (descriptions of integrity constraints on data) should also receive R&D attention. Predicates can be used to validate more than just program logic. They can be attached to data to describe constraints, used to test for the commencement of execution, or attached to documentation fragments. The full power of the use of

predicates throughout a system's description, construction, and maintenance phases has only begun to be exploited.

The relative utilities and weaknesses of the predicate approach, the dataflow approach, the functional approach, and traditional operational techniques need to be studied. For what design aspects of what system types should each be used?

This thrust is related to formal verification, rigorous documentation, transformation from informal to formal requirements, design publication and code skeleton use.

The greatest improvement potential exists in the areas of increased reliability and confidence in software. The net cost savings is estimated to be low-to-moderate.

#### Some Relevant Research and Products

The programming and programming languages research in the 70's revolved about the predicate approach; names such as Hoare, Dijkstra, and Wirth come to mind. Work on formal verification is also related.

Textbooks are available to translate the predicate approach into practice. Two examples are Software Development - A Rigorous Approach, by Jones, and Structured Programming, by Linger, Mills, and Witt. A new programming language, "Prolog" (in Kowalski's Logic for Problem Solving), largely consists of predicates written to be satisfied, rather than procedures written to be executed. It provides an interesting alternative view of the problem-solving process.

### Remarks on Rationale

This approach has been one of the most important wellsprings of improvement in programming during the last dozen years, and it should not be ignored as a producer of additional, useful results.

### References

Brown, F. L., and M. Broy, eds. Program Construction. Berlin: Springer-Verlag, 1979.

Jones, C. B. Software Development: A Rigorous Approach. London: Prentice-Hall, 1980.

Kowalski, R. Logic for Problem Solving. New York: North Holland, 1979.

Linger, R. C., et al. Structured Programming: Theory and Practice. Cambridge, MA: Addison-Wesley, 1979.

Yeh, R. T., ed. Current Trends in Programming Methodology. vols. 1 and 2. Englewood Cliffs, NJ: Prentice-Hall, 1977.

#### A.1.4.4 Exception Handling.

##### Description

Exception handling has been a problem in very high level languages (VHLL), and other attempts to construct software using a high level of abstraction. Both the theory and practice of handling exceptions or unusual conditions needs advancement. Simplicity and a high level of abstraction, goals for most attempts to reduce complexity and increase productivity, make it difficult to deal with detail and exceptions. The aim of this thrust is to decrease that difficulty.

Research directions include investigation of: (1) the types of errors that may safely be ignored, (2) the use of implicit mechanisms for handling errors, (3) the problem of obscure computational flow due to high-level, but poor conceptualization, (4) the incorporation of exception handling into main control flow instead of into a separate program section, (5) the use of interactive "conversations" to define program details, (6) the incorporation of computing and application knowledge into tools, (7) the problems in constructing layers of abstraction within a VHLL, and (8) the benefits of explicitly confining a VHLL context to a restricted problem domain.

This thrust shares benefits with very high level languages, user-oriented requirements language/interface, transformation from informal to formal requirements, and forgiving systems.

Major increases in the usefulness of very high level languages and other high-level tools would occur as a result of the success of this effort. Most of the impact of this thrust will therefore be indirect.

### Some Relevant Research and Products

Exception handling in programming languages has been an active area of discussion and controversy. Many newer languages have devoted particular attention to the feature; for example, Ada has an exception handling mechanism. The extent to which exception handling convenience outweighs the resulting obscurity of the computational flow is unclear, however.

The concept of levels of abstraction has been widely used since the late 60's. Appropriate methods for moving between levels of abstraction to facilitate handling exceptional details, however, have not been widely discussed. The whole concept of fitting exceptions into the problem solving process has received only limited attention.

### Remarks on Rationale

Exception handling is a key stumbling block in the path to wide use of very high level languages and tools, and it may be a key cognitive strategy question as well. This is a thrust with significant potential for advances in both practice and theory.

### References

- Goodenough, J. B. "Exception Handling Design Issues." Sigplan Notices vol. 10, no. 7 (July 1975), pp. 41-45.
- Goodenough, J. B. "Exception Handling: Issues and a Proposed Notation." Commun. ACM vol. 18, no. 12 (Dec. 1975), pp. 683-696.
- Hammer, M. and G. Ruth. "Automating the Software Development Process." In Research Directions in Software Technology, P. Wegner, ed., pp. 784-785. Cambridge, MA: MIT Press, 1979.

Liskov, B., and A. Snyder. "Exception Handling in CLU." IEEE Trans. Software Eng. vol. SE-5, no. 6 (Nov. 1979), pp. 547-558.

Luckham, D. C., and W. Polak. "Ada Exception Handling: An Axiomatic Approach." ACM Trans. Programming Languages vol. 2, no. 2 (Apr. 1980), pp. 225-233.

Winograd, T. "Beyond Programming Languages." Commun. ACM vol. 22, no. 7 (July 1979), pp. 391-401.

#### A.1.4.5 Distributed Functions and Resources.

##### Description

Significant improvement in cost-effectiveness and performance can be achieved by assigning system functions and resources to appropriate processors in a computer network, and, within processors, to software, firmware (microcode), or hardware. Thrusts in this area will produce techniques for discovering software functions to be implemented in higher-performance media, for evaluating the costs and benefits of doing so, and for designing high-performance systems using asynchronously operating, intelligent components. The goal is innovative designs for embedded computer systems that avoid pitfalls previously encountered in single-processor computer systems.

Functions can be implemented in software, firmware, or hardware. Increased performance of firmware over software, and of hardware over firmware, must be balanced against increased circuitry costs and reduced flexibility to accommodate changes; however, reported efforts to build VLSI compilers that produce masks for chips from computer programs may moderate the costs and risks of compiling directly into hardware. Simulations and performance measurements on prototype systems can establish the optimum processor and software-firmware mix. Software monitoring tools in prototype systems can help to identify functions requiring implementation in faster media to obtain necessary performance levels.

Solutions need not be confined to single processors. Networks of cooperating special-purpose and general-purpose processors will be used to construct high-performance, special-purpose systems. Modern microprocessors and anticipated microprocessor developments enhance this approach when tasks are essentially composed of independent,

concurrent or sequential parts. Simulation and prototype performance measurement can help to determine the required capacities of each stage's components. A stage can be implemented by a special-purpose processor or by a general-purpose microprocessor, depending on the capability required.

Microcomputer networks can be used to reduce or increase redundancy among several, otherwise independent embedded systems. Intelligent peripherals (e.g. peripherals controlled by microprocessors) can be shared among network users to reduce redundancy, or can be used as back-ups for local peripherals to increase redundancy and survivability, as desired. Standard (hardware and software) local network interfaces need to be carefully studied, and the requisite operating system functions incorporated into each network host.

Different combinations of software, firmware, and hardware in single processors or in multi-processor networks can satisfy a wide range of performance and size requirements. To the extent that standard microprocessors or special-purpose processors can be used, the need to program and support a variety of computer systems can be reduced. Networks can also reduce system cost and size by allowing functions to be shared among independent systems.

#### Some Relevant Research and Products

Techniques for locating hardware and software bottlenecks are well known; work on high-level microprogramming languages and VLSI compilers is intense; development and standardization of local networks and communication protocols is proceeding rapidly; and R&D projects to build prototype special-purpose multi-microprocessor systems are underway.

A number of research projects are currently underway. There have been several studies of possible uses for sixteen-bit and

thirty-two bit microprocessors in computer networks. The California Institute of Technology has developed a "silicon compiler" that generates chip layouts from functional descriptions, allowing specialized chips to be built rapidly. The MITRE Corporation has a project to construct a secure packet switch using multiple, identical, pipelined microprocessors; and IBM has reported on hardware/firmware/software trade-offs in the design of their 4300 series processors. A bus interface standard exists in Mil-Std 1750.

#### References

Berglass, G., C. Hisgen, and E. Siegel, Multi-Microprocessor Designs for a Secure Packet Switch, MITRE Corp., McLean, VA, tech. report MTR-81W00086, Apr. 1981.

Chu, W. W. "Special Issue on Distributed Processing Systems." IEEE Trans. Computers vol. C-29, no. 12 (Dec. 1980), pp. 1037-1163.

Kleinsteiniber, J. "IBM 4341 Hardware/Microcode Tradeoffs." In Proceedings: The 13th Annual Microcomputer Workshop. Los Alamitos, CA: IEEE Computer Society, 1980.

Mason, J. F. "VLSI Goes to School." IEEE Spectrum vol. 17, no. 11 (November 1980), pp. 48-52.

Van Dam, A., and J. Stankovic, eds. "Special Issue on Distributed Processing." Computer vol. 11, no. 1 (Jan. 1978), pp. 11-57.

#### A.1.4.6 Suitable Communication Interconnection.

##### Description

A variety of physical and logical communications methods are in use. For example, there are minicomputer buses, local networks, and long distance networks, as well as semaphores, monitors, and rendezvous mechanisms. An overall theory is needed to provide a means for selecting, constructing, and modifying communication interconnections.

The Internet Protocol (IP) at the frame level and the Transmission Control Protocol (TCP) at the packet level are the DoD standard low-level protocols. Higher-level protocols need to be defined and standardized so that disparate devices and independently written applications can communicate. When these efforts are successful, hosts, and users having terminals with different capabilities, will be able to communicate. (This will simplify the procurement of different types of terminals.) Applications will be able to receive data from other, unknown applications and transfer data to them. This will facilitate the construction of software from standard components.

To realize the full benefits of computer networks, devices, applications, terminals, and users must be able to communicate with one another. Efforts in this area will build on the standard Internet and TCP protocols to define and implement virtual terminal protocols, file transfer protocols, and command language protocols.

A virtual terminal protocol (VTP) is a mechanism through which a logical terminal communicates with another logical terminal, an operating system, or an application. A logical terminal can be an interface between a virtual terminal and a class of physical termi-

nals, or it can be an interface between a virtual terminal and an application's conception of a physical terminal. A VTP uses an abstraction of a real terminal, called a virtual terminal, which has a well-defined architecture, character set, and function repertoire. Logical terminals map the characteristics of physical terminals to and from the virtual terminal architecture, using the VTP. With this approach, each device needs to communicate with only one other device (i.e. the virtual terminal) to be able to communicate with all other devices in the system.

One approach to inter-application communication is via file transfer. This approach has been implemented successfully in the UNIX operating system on an elementary level. Complicated issues involving character set and file format transformations need to be addressed. If standard components are written to take their input from files of specified structure and put their output onto files of specified structure, a standard file transfer protocol will facilitate the combination and recombination of these components in new applications. Standards organizations are interested in this topic but have not defined the problems.

A common command language protocol would facilitate the transfer of applications and human workers between systems. A user would only need to know one command language, mapped by the common protocol into a virtual command language, which the receiver would map into its command language. Command languages are substantial obstacles to the transfer of programs and programmers between different systems.

The computer system of the future will be a local network tied into a larger network. Processes (i.e. users or applications) will be able to "advertise" for resources in the network, compare bids, and accept the best from a resource of appropriate capability with the spare capacity.

There will be a number of benefits in all phase of the life-cycle. Increasing devices' abilities to communicate at high levels will reduce the amount of software that needs to be rewritten for different environments; virtual terminal protocols will simplify terminal-to-terminal communication and terminal-to-host communication; file transfer protocols are one way to achieve the capability for standard software components; and a command language protocol will simplify the transfer of persons and applications to new environments. There is general agreement on the need for these higher level protocols, if not on the specifics. It will be many years before standards can be defined, but it may be easier to accomplish something in the more controlled, more narrowly defined, world of embedded systems, than in the international information processing world. Once standards are defined, if they are good enough, the potential cost and productivity benefits are great.

#### Some Relevant Research and Products

Some research has been done on virtual terminal protocols, but it is premature to look for products. Most efforts are under the aegis of the International Standards Organization (ISO) and the American National Standards Institute (ANSI).

#### References

Green, P. E. Jr. "Special Issue on Computer Network Architectures and Protocols," IEEE Trans. Communications vol. COM-28, no. 4 (April, 1980).

Hill, I. D., and B. L. Meek, eds. Programming Language Standardisation. Chichester, U.K.: John Wiley & Sons, 1980 (particularly Chapter 11, "Operating Systems Command Languages").

Kahn, R. E., ed. "Special Issue on Packet Communication Networks."  
Proceedings of the IEEE vol. 66, no. 11 (November, 1978).

### A.1.5 Programming

#### A.1.5.1 Transform Software to Improve Quality.

##### Description

Given a useful set of software quality metrics, transformations could be performed on existing software to improve the qualities measured. This would be useful for reliability and maintainability characteristics enhancement.

Algorithms and heuristics intended to improve the structure of code or to turn unstructured programs into structured programs have been suggested; these are known as structuring engines. In addition, transformation methods have been published that transform recursive programs into iterative ones and vice versa, and improve program robustness to prevent abnormal termination.

Human intervention will be needed in the quality improvement process. Bad errors probably cannot be handled by the transformation process, and real-world information may have to be supplied. For example, naming new modules constructed by the transformation process can be done more readily by humans than by machines.

Existing systems might save much of their maintenance costs through the use of a transformation tool. However, the cost to apply the tool, particularly the first time, may be substantial. In any case, such a tool will probably only apply to a few widely used high-order languages.

The tool could be useful in training and education, by showing how programs might be improved.

Moderate cost savings are anticipated, primarily in the operations phase of the life cycle with some benefit in the development phase. There are two major R&D difficulties: lack of good metrics and lack of a good library of transformations.

#### Some Relevant Research and Products.

Software metrics is an active area of research, and many metrics currently exist. What is needed is a comprehensive framework for metrics and a consensus on their relative merits.

Transformations research is active, but most of it is focused on program speed or memory usage optimization. Transformations to improve the structure of programs have been suggested, however. In their book Structured Programming, Linger, Mills, and Witt describe an approach for breaking apart a program into prime subprograms that then may be transformed.

#### Remarks on Rationale

If the quality of existing implementations could be significantly improved, either automatically or through a highly productive man/machine team, then very substantial cost and reliability benefits could accrue. The potential for applying the results of this thrust to existing systems, thereby reducing high maintenance costs, is a big plus.

#### References

Balzer, R., and T. E. Cheatham, Jr. "Special Section on Program Transformations." Trans. Software Engr. vol. SE-7, no. 1 (Jan. 1981), pp. 1-39.

Basili, V. R., ed. Models and Metrics for Software Management and Engineering. Los Alamitos, CA: IEEE Computer Society, 1980.

Kernighan, B. W., and P. J. Plauger. The Elements of Programming Style. 2nd ed. New York: McGraw Hill, 1978.

Perlis, A. J., F. G. Sayward, and M. Shaw, eds. Draft Software Metrics Panel Final Report: Papers Presented at the 30 June 1980 Meeting on Software Metrics, Washington, D. C. Yale University, New Haven, research report 182/80, June 1980.

#### A.1.5.2 Formal Verification of Large Systems.

##### Description

Areas related to this activity have been investigated for more than a decade, but the application of the results to large systems has been limited. Formal verification of large systems is a series of transformations beginning with specification, proceeding through design and implementation (coding), and resulting in assurance that the implementation corresponds to its specification. This does not indicate that the system is "correct," or that it will never fail, or that it will never produce incorrect output. Questions concerning compiler and hardware/firmware validation are excluded from this thrust.

When reasonably mature, this would be an important technology because it would be known (with high probability) that the implementation was at least consistent with the stated requirements as embodied in the specification. As ancillary benefits, the methodologies being developed appear to be leading towards the creation of integrated design and implementation tools and towards the development of automated techniques to point up various deficiencies and omissions in specifications. Furthermore, it may be expected that improvements in documentation and reliability will result along with some reduction in maintenance complexity and expenditures.

Moderate cost savings are expected in the requirements and development phases of the lifecycle, with larger benefits in the operations phase.

### Some Relevant Research

Many projects are documented in the open literature. The DoD Computer Security Seminar of November, 1980, provides a good measure of the state of the art. The focus of the seminar was on "trusted" systems; this thrust differs from large system verification only in that certain "security" constraints are added. The three days of presentations showed that the problems are exceptionally difficult, that the work has not progressed rapidly, and that large system verification is far from realization.

### Remarks on Rationale

Testing can only find errors, it cannot prove their absence. Formal verification proves error absence, albeit in the limited sense of agreement with specifications within an idealized environment. Formal verification activities are key generators of increased understanding and new techniques. Even though formal verification of large systems is not yet possible, semi-formal methods derived from this work may be useful.

The DoD Security Initiative is active in this area; this thrust will provide additional support for reasons other than security.

### References

- ACM. "Workshop on Formal Verification." ACM Software Engineering Notes vol. 5, no. 3 (July 1980), pp. 4-47.
- Chehey1, M. H., M. Gasser, G. A. Huff, J. K. Millen. Secure System Specification and Verification: Survey of Methodologies. MITRE Corp., Bedford, MA, tech. report MTR-3904, Feb. 1980.
- Kreig-Bruckner, B. and D. C. Luckham. "ANNA: Towards a Language for Annotating Ada Programs." ACM Sigplan Notices vol. 15, no. 11

(November 1980), pp. 128-138.

Linger, R. C., et al. Structured Programming: Theory and Practice.  
Cambridge, MA: Addison-Wesley, 1979.

Yeh, R. T., ed. Current Trends in Programming Methodology. vol. 2.  
Englewood Cliffs, NJ: Prentice-Hall, 1977.

#### A.1.6 Testing

##### A.1.6.1 High-Confidence Software Testing.

###### Description

Despite the increased importance and use of other software verification and validation (V&V) techniques, testing is and will remain for a considerable time an important part of V&V activities. Methodologies leading to high confidence through testing are important. It would be useful to know quantitatively what level of confidence is proper both for new systems and for existing systems that have been modified. The keys to this problem are test data generation methodologies, reviews of large volumes of output for correctness, and the development of an understanding of probabilities applied to software testing and reliability. This thrust is aimed at "correctness" and performance, with the goal of producing justifiably high confidence in software.

Various measures of coverage (i.e. fraction exercised) related to underlying domains (such as input data, program code, function types performed, classes of output, potential interactions, requirements, and past errors in similar products) are available for systematically constructing tests and quantifying the results. Available theory concerning the amount and type of test data needed to verify some software functions exists and could be developed to cover others. The objectives of improved quantitative coverage measures and the development of a set of testing theories for individual functions covering much of the relevant software provide fruitful areas for R&D.

These or other systematic approaches to testing that result in high or quantifiable levels of confidence will impact software quality and allow better planning and management of testing. A method of high confidence testing will provide a rational basis for the testing aspect of the government's acceptance decisions for contractor-produced software, thereby benefiting the entire relationship.

The developed methodology could also result in tools, e.g. output comparers/scorers, and test data generated automatically from requirements and designs.

There is potential for improvement from accurate knowledge of the level of risk involved in the use of software. Improvements are also possible in the planning, management, and process of testing. Reflecting the importance of testing, the expected savings are moderate for both new and existing systems.

#### Some Relevant Research/Products

Code coverage measurement tools, called profilers, are available, and some are included in commercial compilers; for example, IBM 360/370 OS/VS COBOL. Other coverage methods and measurements have been developed.

Theories for test data to test linear comparison relations, linear transformations, and sorts have recently been developed. The mutation approach (systematic perturbation of code to see if test data reveals the change) has led to progress in developing test data patterns that reveal likely errors. Information has been collected on common errors in some data processing functions.

The concepts of testing to violate assertions and redundant software could prove fruitful.

### Remarks on Rationale

A number of recent developments in testing allowing systematic and rigorous approaches make this a promising area. Testing is an existing activity in which improvement should be fairly readily accepted.

### References

Chandrasekaran, B., ed. "Special Collections from Workshop on Software Testing and Test Documentation, 1978." IEEE Trans. Software Eng. vol. SE-6, no. 3 (May 1980), pp. 233-290.

Deutsch, M. S. "Tutorial Series 7: Software Project Verification & Validation." Computer vol. 14, no. 4 (Apr. 1981), pp. 54-70.

Howden, W. E. "Completeness Criteria for Testing Elementary Program Functions." In Fifth International Conference on Software Engineering, pp. 235-243. Los Alamitos, CA: IEEE Computer Society, 1981.

Miller, E., ed. "Special Issue on Software Quality Assurance." Computer vol. 12, no. 8 (Aug. 1979), pp. 7-42.

Miller, E., et al. "Workshop Report: Software Testing and Test Documentation." Computer vol. 12, no. 3 (Mar. 1979), pp. 98-107.

Miller, E., and W. E. Howden, eds. Tutorial: Software Testing and Validation Techniques. Los Alamitos, CA: IEEE Computer Society, 1978.

Myers, G. J. "A Controlled Experiment in Program Testing and Code Walkthroughs / Inspections." Commun. ACM vol. 21, no. 9 (Sept. 1978), pp. 760-768.

Myers, G. J. The Art of Software Testing. New York: Wiley-Interscience, 1979.

### A.1.7 Operations

#### A.1.7.1 Facilitating System Evolution.

##### Description

Much of the software of interest to DoD is within evolving long-lived, large systems. Planning for and handling changes throughout a system's evolution will be difficult, requiring provisions for known future changes as well as for unexpected changes.

Much work remains to be done to facilitate the evolution of large systems. Research usually focuses on one facet of the total problem, thereby failing to uncover results that could aid the entire lifecycle. The goal of this thrust is to examine the entire life of software.

Research is needed on: the environment in which software systems exist; the ways in which the need for changes becomes evident; the methods by which changes are, and should be, proposed, evaluated, implemented, tested, and accepted into the system; and the relationship between software and the hardware on which it operates. Longitudinal studies following large systems throughout their lives and intensive studies of the human factors involved in system design and maintenance will be needed.

Moderate cost benefits will accrue in the operations phase of the lifecycle due to improvements in handling system changes, as a result of this thrust. Some minor cost benefits will probably also appear in the development phase. This thrust could also benefit existing systems, although it is not clear to what extent the results would be readily applicable.

### Some Relevant Research and Products

Software maintenance is just beginning to receive serious research attention, despite the large sums being spent on it. The Navy Research Laboratory has an experiment underway to redo the avionics software requirements and design for the A-7 to emphasize the capability for ease of change. While ease of modification is one frequently stated motivation for many modern software techniques, actual research on modification is almost nonexistent. The main tools in use are aimed at software configuration maintenance and control.

### Remarks on Rationale

Ongoing, broadly-based research on the evolution of large systems is needed now and will be needed in the future as systems and system design methods continue to evolve. As it is quite difficult to perform accurate retrospective studies, research is needed to examine the evolution of systems currently under development, efforts that must continue over the many years of data collection that will be needed. The work must start now to build the data base that future researchers will depend on in their search for ways to obtain major improvements in the field of software engineering.

### References

Donahoo, J. D., and D. Swearingen. A Review of Software Maintenance Technology. Rome Air Development Center, Griffiss AFB, NY, tech. report RADC-TR-80-13, Feb. 1980.

Henniger, K. L. "Specifying Software Requirements for Complex Systems: New Techniques and Their Applications." IEEE Trans. Software

Eng. vol. SE-6, no. 1 (Jan. 1980), pp. 2-13.

Lientz, B. P., and E. B. Swanson. Software Maintenance Management.  
Reading, MA: Addison Wesley, 1980.

#### A.1.7.2 Impact Analysis of Proposed Change.

##### Description

Given that a certain (software, hardware, operating system) modification is to be made, then an automated answer is needed to establish what will require changing or investigation for potential change. This would include not only code but all aspects, including documentation, requirements, test data, and personnel.

Sometimes "minor" system changes are made that result in major headaches or system failure; at other times "major" changes are not made, because of the fear of such headaches, when the change would actually be small and safe. What is needed is an automated method for determining the ramifications of a proposed change within a software system. Static examination of code modules could reveal the effects of changes to code elements and module structures. Current impact analysis practices are generally poor, usually depending on the judgment of a programmer with no time to track down all of the necessary changes, or unaware of some of them.

Impact analysis is critical, however, if management is to control the update and maintenance process. Calculation of cost/benefit factors depends on accurate determination of the costs; accurate cost determinations are difficult to make manually when many proposed changes must be examined.

This research would attempt to determine the best methods for performing an automated impact analysis and the scope that such an analysis should have if it is to be practical. It would also define the data needed from other systems in the integrated software support environment.

Major benefits will accrue from the development of a dependable automated impact analysis mechanism. It will become easier to predict the effort involved in making changes to system code, because the estimator will have better knowledge of the changes to be made. The probability of incorrect or incomplete alterations will be reduced, and the efficiency of the programmer making the changes will be higher. The productivity of maintenance programmers will increase markedly. Estimates of task sizes will be made more accurately, more completely, and in less time. Moderate cost savings will result from implementation of a system designed to examine high-level language modules.

#### Some Relevant Research and Products

Software Research Associates of San Francisco, California, markets an Interactive Semantic Update System (ISUS) for FORTRAN that performs static analysis of changes to FORTRAN code.

#### References

Software Research Associates. Examples of Interactive Semantic Update System (ISUS) Use. Technical Note TN-749/2, March, 1981.

Weiser, M. "Program Slicing." In Fifth International Conference on Software Engineering, pp. 439-449. Los Alamitos, CA: IEEE Computer Society, 1981. Yau, S. S. Self-Metric Software. 3 vols. Rome Air Development Center, Griffiss AFB, NY, tech. report RADC-TR-80-138, Apr. 1980. (NTIS accession nos. AD-A086 290 thru 292.)

Yau, S. S., and J. S. Collofello. Performance Ripple Effect Analysis for Large-Scale Software Maintenance. Rome Air Development Center, Griffiss AFB, NY, tech report RADC-TR-80-55, Mar. 1980. (NTIS accession no. AD-A084 351.)

Yau, S. S., and J. S. Collofello. "Some Stability Measures for Software Maintenance." IEEE Trans. Software Eng. vol. SE-6, no. 6 (Nov. 1980), pp. 545-552.

## A.2 Managerial

### A.2.1 Acquisition Manager's Support System

#### Description

The specification and acquisition of software is complicated, requiring technical, contractual, and managerial skills. A knowledge-based acquisition system will provide the breadth of knowledge required, as well as additional assistance, review, and discipline. Contracting officers, their technical representatives, weapons systems program managers, and software subsystem managers would benefit from this assistance.

A manager's support system should cover the full acquisition process from the initial problem statement and request for proposals (RFP) through the software system lifecycle. The acquisition manager must be familiar with existing as well as proposed DoD, Federal, and particular service or agency procurement and lifecycle regulations, directives, instructions and circulars. The thrust on improving the acquisition process (making it more consistent with the realities of software) should help this effort.

The manager's support system would enable the manager to request complete documentation or specific information. The system should help the manager: 1) formulate a technical statement of work for a proposal based on a problem statement, 2) monitor the RFP process, 3) evaluate proposals for meeting technical, cost, personnel and management requirements, and 4) monitor the status and progress of the

contractor's activities for the lifecycle of the system being acquired.

Knowledge-based system techniques could be applied to develop the data bases and procedures required to demonstrate this support system. The initial support system could be evaluated by use in a DoD agency before propagation throughout DoD.

A major benefit of this support system would be to improve the performance of acquisition management and contracting. Another major benefit would be to decrease the acquisition process time, and therefore, to shorten the time from problem statement formulation to contractor selection and actual software system operational use. The net estimated expected savings is moderate to high.

#### Some Relevant Research and Products

Using the results of the thrust on improving the acquisition process, government documentation, and knowledge of past procedures to modify or redesign techniques and procedures for the acquisition process will require considerable insight and creativity. Few new technological breakthroughs are required although results of ongoing developments in software cost estimating requirements and specification languages would be beneficial. This R&D effort may also result in fusion with management tools with which an acquisition manager should be knowledgeable.

#### Remarks on Rationale

Comments are usually made that it requires years from the time a problem is stated before a contractor is selected to work on a proposed solution. Also, estimates of cost and time required to develop a proposed solution are usually low. The acquisition manager should have a support system that will decrease the time required for

contracting (recognizing that the decision process may still take considerable time, because of the number of people involved) and provide more accurate estimates of cost and time for the acquisition process.

### References

Digman, L. A., and G. I. Green. "A Framework for Evaluating Network Planning and Control Techniques." Research Management (Jan. 1981), pp. 10-17.

Jones, V. E., chairman. Final Report of the Software Acquisition and Development Working Group. Washington, DC: ASD for C3I, July 1980.

Logicon Technical Staff. Management Guide to Avionics Software Acquisition. 4 vols. Aeronautical Systems Division, Wright-Patterson AFB, OH, tech. report ASD-TR-76-11, June 1976. (NTIS accession nos. AD-A030 591 thru 594.)

McCosh, A. M., and M. S. Scott-Morton. Management Decision Support Systems. New York: Wiley, Halstead Press, 1978.

Merwin, R. E., ed. "Special Section on Software Management." IEEE Trans. Software Eng. vol. 4, no. 4 (July 1978), pp. 307-361.

Mish, R. K. Software Acquisition Management Guidebook: Series Overview. Electronic Systems Division, Hanscom AFB, MA, tech. report ESD-TR-78-141, March 1978. (NTIS accession no. AD-A055 575.)

Nunes, S. E. "Engineering Management in Government Contract Work." Computer vol. 14, no. 2 (Feb. 1981), pp. 86-87.

Putnam, L. H., ed. Tutorial: Software Cost Estimating and Lifecycle Control. Los Alamitos, CA: IEEE Computer Society, 1980.

Thayer, R. N., A. Pyster, and R. C. Wood. "The Challenge of Software

Engineering Project Management." Computer vol. 13, no. 8 (Aug. 1980), pp. 51-59.

Winston, P. M. Artificial Intelligence. Menlo Park, CA: Addison Wesley, 1977.

SAE Guidebooks - Application and Use, ASD USAF, Wright-Patterson AFB, Ohio, ASD-TR-80-5028, 1980.

### A.2.2 Software Technology-Compatible Acquisition

#### Description

While progress has been made in DoD's ability to contract for software, a number of technological truths concerning software are not always reflected in the acquisition process.

Many technical attributes of software and its development/maintenance have potential impact on and implications for the DoD acquisition process. These attributes include (but are not limited to): (1) small, elite programmer teams can produce more than large, mediocre teams; (2) correct requirements are difficult to establish initially, resulting in significant risk plus much "development" performed as maintenance; (3) changes usually are frequent; (4) simplicity, elegance, and rigor are essential; (5) software is often the highest risk in a new system; and (6) the same software may have many representations (source code, object code, design documentation, users' manual, etc.) and may exist in several versions (releases, configuration dependencies, and different "fixes" applied).

The relevant aspects of software need to be identified, and alternative approaches to acquisition need to be generated and evaluated in this light. Pilot efforts at procurement will be conducted to explore and validate approaches. Publication of model contracts will spread the developed practices.

Potential for improvement exists in the decreased frequency and importance of surprises and in the reduction of conflict among the organizations and persons involved. Improvement should also occur in the utility and maintainability of the systems produced. The potential cost savings over the lifecycle would be large if the effect of

such an acquisition process were to cause program managers and contractors to follow its guidelines faithfully. The chance of R&D success is high. The technology transfer percentage is fair, and the net estimated expected savings is low to moderate.

#### Some Research Results or Products

Software contracts in the commercial sector also have a mixed record; however, lessons can be learned from the better practitioners. Non-contractual relationships that successful in-house software shops have with their users/buyers also provide clues to a successful acquisition methodology.

Many technological truths are known concerning software. Using these truths as a basis for modification (redesign) of the procurement/contracting process will require insight and creativity. Few new technological breakthroughs are needed, though progress in areas such as software metrics would clearly be beneficial. The recent C3I working group on software development and acquisition concentrated its recommendations on acquisition contracting issues.

#### Remarks on Rationale

It is not unusual to hear remarks such as, "We know how to build software; we just do not know how to buy it." While the first part of the remark may be questionable, the second appears to be true. As we learn how to build and maintain software, our acquisition process should change to facilitate good practices.

### References

Glass, R. L. "The Importance of the Individual." ACM Software Engineering Notes vol. 5, no. 3 (July 1980), pp. 48-50.

Jones, V. E., chairman. Final Report of the Software Acquisition and Development Working Group. Washington, DC: ASD for C3I, July 1980.

Mish, R. K. Software Acquisition Management Guidebook: Series Overview. Electronic Systems Division, Hanscom AFB, MA, tech. report ESD-TR-78-141, March 1978. (NTIS accession no. AD-A055 575.)

Myers, W. "A Statistical Approach to Scheduling Software Development." Computer vol. 11, no. 12 (Dec. 1978), pp. 23-35.

Scharer, L. "Pinpointing Requirements." Datamation vol. 27, no. 4 (Apr. 1981), pp. 139-151.

Walston, C. E., and C. P. Felix. "A Method of Programming Measurement and Estimation." IBM Systems Journal vol. 16, no. 1 (1977), pp. 54-73.

SAE Guidebooks - Application and Use, ASD USAF, Wright-Patterson AFB, Ohio, ASD-TR-80-5028, 1980.

### A.2.3 Technology Transfer in the Software Area

#### Description

Improved practices often exist in research efforts and industry before they are used widely within the DoD community. When DoD's practices move closer to and remain near the state of the art in software technology, performance will be significantly upgraded.

Areas with good technology transfer in DoD should be examined, and barriers to software technology transfer identified. Strategies and mechanisms for locating new technology, evaluating it, and transferring it appropriately must be developed, tried out, and improved until they function satisfactorily. Strategies will address such issues as how to identify technologies worthy of transfer, how much to spend on implementing transfer, who must be informed/influenced in order to have certain types of technology successfully adopted, and how to evaluate and improve the technology transfer effort itself.

Mechanisms for communicating and training need to be addressed. More widespread use of newsletters, technique monographs, audiovisual courses, correspondence courses, schools/training centers, and traveling seminars focused on dissemination of advanced software technology will be helpful.

While good technology transfer is a prerequisite for success by all the other STI thrusts, the benefits estimated here are only for transfer of technology already in existence. Net savings will be moderate to high.

### Some Relevant Research and Products

Technology transfer studies and efforts have been performed for DoD on other topics. Because of the rapid pace of technological innovation, most of the software industry has problems similar to those of DoD. Many courses, both live and recorded, already exist. In addition, the educational establishment continues to produce some relevant textbooks and courses. Some of the work on technology transfer export restriction may also be relevant.

### Remarks on Rationale

Technology not deployed and utilized has little value. The rapid transfer of R&D results into use can provide a significant improvement in performance.

Good software technology transfer would provide relatively inexpensive benefits. In addition, effective technology transfer is essential for realizing the full benefit of the STI and of other sponsored R&D.

### References

Auerbach Publishers. Auerbach Information Management Series. Pennsauken NJ: Auerbach.

COMTEC-2000 Study Group. Computer Technology Forecast and Weapon Systems Impact Study (COMTEC-2000). 3 vols. HQ Air Force Systems Command, Washington, DC, tech. report 78-03, Dec. 1978 - July 1979, vol. 3, ch. 6 ("Technology Exploitation Time Lag").

Goodman, S. E., chairman, N. S. Glick, W. K. McHenry, et al. Software Technology Transfer and Export Control. Arlington, VA:

Institute for Defense Analysis, July 31, 1980.

Martens, J., and L. Duvall. "The Role of an Information Analysis Center in Software Engineering Technology Transfer." In 1980 National Computer Conference, AFIPS Conference Proceedings, vol. 49, pp. 677-682. Arlington, VA: AFIPS Press, 1980.

Squires, S., chairman. DoD Workshop on Software Technology, 15-16 May 1980, Fort Belvoir, VA: OUSDRE, 1980.

### A.3 Personnel-Related

#### A.3.1 Superperformer Competencies

##### Description

A better understanding of the traits and behavior of individuals who are superperformers, i.e. those who are recognized as being an order of magnitude better than others in some aspect of software development and maintenance, is sorely needed. Research has shown great variation among the achievement rates of similarly-trained software technicians. High achievers and comparable sets of average and possibly low achievers should be studied through intensive interviews and other techniques to identify causative behaviors and characteristics. Such understanding will help to select or assign individuals to software specialties.

Research and development can then be directed at the transfer of these behaviors and characteristics to others. Some of these behaviors may already be taught in the better academic programs, but may be scarce in the pool of practitioners. Experiments will be conducted on the success of transferring behaviors by formal training or other educational methods.

The following questions will need to be answered. Can the behaviors and characteristics be transferred? If so, how can this best be accomplished? Once transferred, do these acquired behaviors and characteristics result in the expected improvements in productivity?

Effective and usable technology transfer/training packages need to be developed for the various software technician and management roles and levels. These might be utilized in intensive advanced training for programmers or in other DoD training and professional development efforts.

This thrust is related to the improvement of technology transfer, which aims at introducing and disseminating the best industry practices. The potential benefits are enormous and potentially span most of the significant issues in software; however, it would be unrealistic to expect an across-the-board substantial improvement. Areas with heavy user/buyer involvement might be less impacted.

The chance of research success is hard to judge. Even if some abilities are hard to identify and transfer, others probably will be reasonably easy.

While technology transfer penetration percentage will depend, in part, on the mechanisms used and the funds expended, it is expected that in the end the percentage will be fairly high. If 10% of the staff were to become ten times better, then they would approach the performance of the entire prior staff. This may take some time, however, since the synergistic effect of better people building better software, which is then easier for skilled people to maintain, will require a number of years to propagate throughout DoD. Original development as well as maintenance changes should occur more rapidly once this capability is achieved. The smaller number of software professionals required should also ease planning and control difficulties.

### Some Relevant Research and Products

Robert Glass surveyed studies showing large individual differences and proposed such a thrust as this. Psychologists have studied a number of occupations/positions using the comparison of good and mediocre performers to derive competencies that can be taught. For example, MacBer has done a number of such studies for the Navy. Unfortunately little rigorous validation of the success of any of these efforts has been done.

### Remarks on Rationale

This is one of the few areas where research results show order-of-magnitude differences to be exploited. The potential benefit greatly exceeds the limited investment required.

### References

Computing Trends. A Summary of Capabilities and Experience. Seattle, WA: 30 July 1980.

Glass, R. L. "The Importance of the Individual." ACM Software Engineering Notes vol. 5, no. 3 (July 1980), pp. 48-50.

Goldman, D. "The New Competency Test: Matching the Right People to the Right Job." Psychology Today Jan. 1981, pp. 34ff.

Mazlack, L. J. "Identifying Potential to Acquire Programming Skill." Commun. ACM vol. 23, no. 1 (Jan. 1980), pp. 14-16.

### A.3.2 Intensive Advanced Programmer Training

#### Description

A training process that transforms moderately inexperienced programmers into experts will greatly improve productivity and quality in software. As in Army ranger training, indoctrination, training, and realistic exercises are needed to establish a capability for modern programming.

Because of the rapid obsolescence of computer knowledge, it is difficult for a programmer or analyst to build up a body of current, relevant knowledge. Most of what he knows is quickly outdated, and a single individual rarely has knowledge of many relevant projects from start to finish. College courses are necessary to form a knowledge base, but they cannot provide sufficient experience for the real world of large, military systems.

After a programmer or analyst has gained real-world knowledge, it will be useful for him to participate in an advanced training program. The training would not necessarily take place in classrooms; it could be conducted through a national network such as the ARPANET or AUTODIN II. Assignments could be presented and worked on, and small programming teams could interact over the network.

The courses would be directed toward military, real-time problems, and will enable the students to interact and use one another's experiences as resources. The use of realistic exercises will enable the students to learn through experience.

This effort would design courses for important areas and evaluation methods to measure the effectiveness of such courses.

Major benefits could accrue to DoD through establishment of an advanced series of courses of proven value. Better training should directly impact productivity.

#### Some Relevant Research and Products

Intensive "summer" schools like those given at MIT and the University of California at Santa Cruz are good models. NATO-sponsored summer schools on software and the Naval Research Laboratory summer courses on software engineering are also interesting examples. Some educational efforts in private industry are relevant, for example the IBM course described in the book by Jones. Related, but on a less intense level, are new Master's degree programs in software engineering. Examples of these are the programs offered by the University of Seattle and the Wang Institute.

DoD has a substantial history of success with intensive training in other subjects, e.g. Ranger training and pilot training. The principles and techniques used are also relevant here.

#### Remarks on Rationale

Several companies have found in-house advanced training for programmers to be cost-effective. Because DoD has requirements that are not taught in traditional computing courses (e.g. real-time operation, security and survivability, high criticality, and unusual mathematical operations), specialized courses in these areas are required.

### References

Chumura, L. J. et al. Software Engineering Principles. Washington, DC: Naval Research Lab, July 1980. (NTIS accession no. AD-A087 997.)

Fasang, P. P., and D. C. Rine. "Computer Science and Engineering Curricula: the Bridge from Theory to Applications." Computer vol. 13, no. 6 (June 1980), pp. 37-42.

Jones, C. B. Software Development: A Rigorous Approach. London: Prentice-Hall, 1980.

Mulder, M. C., ed. "Special Supplement: Computer Science and Engineering Education." Computer vol. 10, no. 12 (Dec. 1977), pp. 70-133.

Ramamoorthy, C. V. "Computer Science and Engineering Education." IEEE Trans. Computers vol. C-25, no. 12 (Dec. 1976), pp. 1200-1206.

### A.3.3 Programmer Laboratory

#### Description

A laboratory to study how work is actually performed by software practitioners (behavior, interactions, cognitive techniques) could facilitate the development of tools and techniques leading to increased productivity in software creation, use, and maintenance. A central laboratory would be useful for researchers setting up experimental software packages. Experiments in group dynamics and in the influence of physical surroundings could be performed. The laboratory environment would also help to ensure that changes in worker productivity are not due to uncontrolled, external events at remote work locations. In addition, new techniques could be demonstrated and instructors trained in new methods soon after the new methods are validated and codified. Researchers would train the first group of instructors, who would then leave to set up standard courses at their own training centers.

The laboratory need not be restricted to one physical location; part of the facilities could be a set of specialized software available on a nation-wide network. The accessibility of the software would encourage its use in real-life situations, giving results to complement those from isolated laboratory experiments.

The primary benefit of the programmer laboratory is that it will support several other thrusts. The central laboratory will make the researcher's job easier by providing equipment and staff when needed, and it will encourage the interaction of researchers working on different tasks who might otherwise not meet. Benefits will accrue during all phases of the lifecycle. The cost savings of a lab already

set up and usable by many different researchers will be considerable, and the savings from a methodology developed by a number of researchers meeting the lab may be huge.

#### Some Relevant Research and Products

The University of Maryland, NASA, and Computer Sciences Corporation have an effort called the Software Engineering Laboratory underway at the Goddard Space Flight Center. This contractor software shop is monitored as it carries out its normal operations. Much of the data is collected from forms filled out by staff members who report efforts expended on each project. Emphasis to date appears to be on software costing and complexity models, and on metrics.

Psychological research in the software area is relatively new; a summary of recent work is given by Schneiderman in his book Software Psychology. Programming language features, stylistic differences, and approaches to debugging have been studied. In more traditional psychological/social science areas, environments (physical, social, and management), personality, motivation, and group processes have been studied for software personnel. The value of much of this work is limited, either because students (and small projects) were the subjects, or because industrial studies were inadequately controlled.

#### Remarks on Rationale

A number of other thrusts in the STI require the existence of a programmer laboratory. Development of such a laboratory in a carefully controlled manner, will ensure that it will become a center for a broad spectrum of research activities, bringing together diverse tasks with related objectives.

## References

Basili, V. R., M. V. Zelkowitz, F. E. McGarry, R. W. Reiter, jr., W. F. Truszkowski, and D. L. Weiss. The Software Engineering Laboratory. University of Maryland Computer Science Center, College Park, MD, tech. report TR-535, May 1977.

Black, R. K. E., R. P. Curnow, R. Katz, and M. D. Gray. BCS Software Production Data. Rome Air Development Center, Griffiss AFB, NY, tech. report RADC-TR-77-116, Mar. 1977. (NTIS accession no. AD-A039 852.)

Chrysler, E. "Some Basic Determinants of Computer Programming Productivity." Commun. ACM vol. 21, no. 6 (June 1978), pp. 472-483.

Curtis, B. "Measurement and Experimentation in Software Engineering." Proceedings of the IEEE vol. 68, no. 9 (Sept. 1980), pp. 1144-1157.

Green, T. R. G. "Programming as a Cognitive Activity." In Human Interaction with Computers, H. T. Smith and T. R. G. Green, eds., pp. 271-320. London: Academic Press, 1980.

Myers, G. J. "A Controlled Experiment in Program Testing and Code Walkthrough/Inspection." Commun. ACM vol. 21, no. 9 (Sept. 1978), pp. 760-768.

Schneiderman, B. Software Psychology: Human Factors in Computer and Information Systems. Cambridge, MA: Winthrop, 1980.

Walston, C. E., and C. P. Felix. "A Method of Programming Measurement and Estimation." IBM Systems Journal vol 16, no. 1 (1977), pp. 54-73.

#### A.3.4 Personnel Independence

##### Description

Within DoD, personnel changes more frequently than hardware; therefore, there is greater need for personnel independence than for configuration independence. Personnel independence can be aided by completely capturing all relevant knowledge about software, and by organizational and transitional strategies both to retain personnel and to transfer responsibilities properly among them. The learnability and understandability of software representations clearly have impacts as well.

Throughout the lifecycle, provisions must be made to minimize the impact of personnel changes. Ideally one should be able to tolerate all potential changes in software personnel. The appropriate level of independence and the techniques to achieve it are the subjects of this activity.

This effort is related to: integrated software support environment, system dictionary/directory, software engineer's support system, and other efforts helping to completely capture software so that new personnel can learn about it.

The initial benefits will be in methods for reducing cost and risk caused by loss of continuity. The principal benefits will accrue during the operations phase of the lifecycle, but some will also accrue during development. The total cost savings will be low on the average; however, risks will be greatly reduced.

### Some Relevant Research and Products

Some standards exist for documentation (DoD and FIPS), and experience has been gained in their usefulness for personnel independence. Some organizations practice a policy of always having at least two people who are familiar with each element of a system.

### Remarks on Rationale

The goal of personnel independence was suggested by the goal of hardware independence.

### References

Couger, D., and R. A. Zawacki. Motivating and Managing Computer Personnel. New York, NY, Wiley, 1980.

Data Management. "Turnover Seen as the Result, Not the Cause of Performance Problems." Data Management vol. 19, no. 4 (April, 1981), p. 22.

McLaughlin, R. A. "That Old Bugaboo, Turnover." Datamation vol. 25, no. 11 (Oct. 1979), pp. 97-101.

Patterson, M. B. "Motivating Your Staff." Data Management vol. 19, no. 4 (April, 1981), pp. 23-30.

#### A.3.5 Improved Education About Software

##### Description

The production of software and software-intensive systems satisfying everyone's needs and expectations is only possible when there is effective feedback between developers and concerned DoD personnel. In order for technically untrained (in software) personnel to be effective, they need to understand enough about software problems and possibilities to ask the right questions and interact constructively. More emphasis is needed in obtaining sufficient general knowledge about the realities of software development to enable requirements developers, specification writers, contract managers, and users to interact effectively with one another and with software developers.

Better informed requirements writers will not demand unachievable reliability or unnecessary performance. Knowledgeable specifiers will not demand the use of inappropriate hardware, software, or procedures. Users who have been able to participate in the early stages of a procurement will not see themselves as its victims.

This thrust will survey existing courses and training materials related to computer (hardware and software) awareness training for non-technical professionals, review DoD's needs in this area, and then create a new, integrated set of courses and materials for DoD use.

### Some Relevant Research and Products

Material and seminars are available that give an introduction to computing, but few teach how to interact with systems and software personnel within the context of systems development and maintenance. Research is being done in the areas of organizational change and interorganizational relations, but only a small part of the literature directly reports on changes caused by computerization.

### References

Huse, E. F. Organization Development and Change. 2nd. ed. St. Paul, MN, West Publishing Co., 1980.

Keen, P. G. W. "Information Systems and Organizational Change." Commun. ACM vol. 24, no. 1 (Jan. 1981), pp. 24-33.

London, K. The People Side of Systems. London, Eng., McGraw-Hill, 1976.

Lucas, H. C. Why Information Systems Fail. New York, NY, Columbia University Press, 1975.

Mumsford, E., and D. Henshall. A Participative Approach to Computer Systems Design. New York, NY, Wiley, 1979.

#### A.3.6 User Programming

##### Description

Because highly skilled software personnel are scarce and expensive, more software development and maintenance tasks should be switched to users. Tasks that are switched must be easy to learn, user-oriented, and have friendly interfaces and tools. Such tools could also be used by low-skilled software personnel. This capability could be particularly attractive to DoD in areas where requirements changes need to be implemented in the field, for example in electronic warfare.

Examples of tools are specialized systems to permit users to perform their own programming after only an hour or two of instruction. A number of query systems have been developed, as have application program generators to produce customized software packages from answers a user gives to a set of questions.

The prediction of likely changes (so that the capability to accomplish them can be built into a system), automatic system changes to reflect changes in user-oriented requirements statements, and an intuitive "programming" interface (based on the principle of directly manipulatable objects) are all questions to be addressed. In the last of these, a graphic representation of a system might be displayed, and then changed by someone touching items on the screen to rearrange them.

Table- or data-driven systems changed by user-supplied control data are steps towards user programming. Designs making "systems so advanced that they are simple" often have facilities for user

changes, and the concept of "unplug one box, plug in another" could be applied to software.

The principal benefit is the elimination of dependence on scarce programmer resources. Other benefits include better understanding of the requirements by the programmer/user and lessened dependence on immediate availability of programmers. Cost savings may not be large; however, this thrust will help alleviate programmer shortage problems and will allow tasks to be undertaken that might otherwise be impractical.

#### Some Relevant Research and Products

High level data base inquiry and reporting facilities are being widely utilized by users to do their own programming. Of particular interest are products, like Query by Example (IBM), with some informality in their user interfaces.

Products, such as statistical packages, which are aimed at particular classes of users, and which deal with users in their language, have had considerable success in eliminating the need for programmers.

#### References

Blasgen, M. W., et al. "System R: An Architectural Overview." IBM Systems Journal vol. 20, no. 1 (1981), pp. 41-62.

Hammer, M., W. G. Howe, V. J. Kruskal, and I. Wladawsky. "A Very High Level Programming Language for Data Processing Applications." Commun. ACM vol. 20, no. 11 (Nov. 1977), pp. 832-840.

Martin, J. Application Development without Programmers. Carnforth, Lancs., U.K.: Savant Institute, 1980.

Reisner, P. "Use of Psychological Experimentation as an Aid to Development of a Query Language." IEEE Trans. Software Eng. vol. SE-3, no. 3 (May 1977), pp. 218-229.

Shoor, R. "Query Users Seek Ease of Use." Computerworld vol. 15, no. 12 (23 March 1981), p. 1.

Zloof, M. M. "Query-by-Example: A Data Base Language." IBM Systems Journal vol. 16, no. 4 (1977), pp. 324-343.

#### A.4 Continuity-Related

##### A.4.1 Voice Replaces Text

###### Description

Many software workers are poor writers with insufficient time to produce good documentation; much software is developed and delivered with inadequate documentation. Technology exists for recording and reproducing voice, and tools exist for editing voice. In addition, technology exists for visual displays to occur simultaneously with voice reproduction.

Effective (if stylized) technology exists to turn text into voice. Within the decade, technology will probably be developed to turn voice into text; some industry observers predict commercial products by 1984. Initially such products will not attempt to understand voice, but will display alternative texts matching the spoken words, permitting the user to select among the alternatives.

Voice and documentation can be combined with a program text display that would include a pointer (such as a cursor) or other highlighting technique. Voice can also be used to augment a program's written description. In addition, demonstrations of a system in action can be accompanied by the author's spoken description. The software's author can be interviewed using a structured interview technique to ensure that all requisite points have been covered. Voice can be used to input program code, and to record comments on software maintenance changes. The problem of indexing voice reference documentation will need to be addressed.

A portion of documentation costs could be eliminated through successful use of the results of this thrust. Most of the benefit would occur in the operations phase of the lifecycle from better documentation, although some would also appear in the development phase.

#### Some Relevant Research and Products

Relevant efforts are in the development of audiovisual training and in computer aided instruction systems that use voice. Voice editors have been developed, for example, in MITRE's TICCIT CAI system.

Some firms have recorded structured interviews with programmers as the basis from which to write documentation. Technical writers can perform the interviews and then listen to the tape, rewriting as appropriate to develop the documentation. Rewriting is minimized by conducting the interview in the same format as the required documentation.

#### Remarks on Rationale

Most programmers appear to be better writers of programming languages than of English. Yet, when two programmers sit down together, one of whom knows the program and one of whom is learning it, the conversation is reasonably successful. What can be done to make the programmer's knowledge available even after he has left? One solution is to have him write down all he knows, with or without the assistance of a technical writer working from interviews with him. Another could be to record what he has to say and keep it in the original voice form. This loses the ability to interact, but should result in much more being recorded than if the information had to be written down.

Voice, and sound in general, has the advantage that it can be combined with displays. Guided tours through the text of the program could be as helpful as recorded guided tours through museums.

#### References

Dahmke, M. C. "Computer Speech: An Update." Byte vol. 6, no. 2 (February 1981), pp. 6-12.

MITRE. An Overview of the TICCIT Program. MITRE Corp., McLean, VA, report M76-44, July 1976.

Verdon, P. "A Closer Look at the TI Speak and Spell." Byte vol. 6, no. 4 (April 1981), pp. 150-154.

#### A.4.2 Built-In Training and Documentation

##### Description

A product with a built-in training capability would train new operators and users without the need for instructors. Instead of operators' and users' manuals, the system would have on-line access to documentation. Simulation modes for play, step-by-step guidance, and help facilities all point in this direction. Systems requiring little training (for example, menu driven) also ease the problem.

Built-in training should allow a user to log on to an operational system and to place the system in a "training support" mode that is functionally the same as the operational mode. The training support mode would allow the user to select or specify the system configuration, the inputs and the outputs. This mode would utilize a "training" data base separate from the operational data base. Built-in training will allow the user to specify the system configuration via menu selections, if desired, or the training mode will simulate the inputs and outputs. The user will be presented with information to explain the training mode, operating instructions and possible system configurations. The training mode should allow the user to be trained by self-help steps through an operational scenario using computer-aided instruction techniques, and to identify areas requiring further training and instruction. In cases where further instruction in the system use is required, appropriate sections of manuals or other documentation should be presented to the user. The training mode will not affect system operation.

Development of built-in training capabilities will decrease user training costs by reducing (in some cases eliminating) the need for

instructors and classrooms and the time for the user to become knowledgeable about the system. Also, for users who have been away from the system for some time, built-in training should refresh their memory quickly on system use. When system changes or upgrades are made, the features can also be easily included in the training mode, resulting in time savings and cost reduction to upgrade users' capabilities.

Many interactive computer systems today have built-in help or instruction-following procedures. The extension of these procedures and techniques to train operational users may be difficult but should be possible. The net estimated savings is expected to be moderate, and to occur in the operational phase of the lifecycle.

#### Some Relevant Research and Products

Few new, if any, technological breakthroughs are required for this R&D effort. Results of on-going developments in computer-aided instruction, man-machine communication, information retrieval, and knowledge-based systems will be beneficial to this thrust.

#### Remarks on Rationale

Training users on new operational systems or when upgrades or significant modifications have been made to their systems is time-consuming and can be costly in terms of travel and time away from the operational site. A built-in training mode will allow a user to be trained on-the-job at his convenience and at his own pace.

### References

Hayes, P., E. Ball, and R. Reddy. "Breaking the Man-Machine Communications Barrier." Computer vol. 14, no. 3 (March 1981), pp. 19-30.

Nievergelt, J. "A Pragmatic Introduction to Courseware Design." Computer vol. 13, no. 9 (September 1980), pp. 7-21.

Nievergelt, J., and J. Weydert. "Sites, Modes, and Trails: Telling the User of an Interactive System Where He Is, What He Can Do, and How to Get Places." In Proc. IFIP Conf. Methodology of Interaction, Seillac, North Holland, 1979.

Salton, G. "Automatic Information Retrieval." Computer vol. 13, no. 9 (September 1980), pp. 41-56.

## B. OTHER IDEAS

This appendix contains very brief descriptions of all ideas currently categorized as unsuitable to be recommended for separate thrusts. They are given here so that reviewers can comment on that categorization. Only the fundamental concepts are described; there is no attempt to portray a proposed effort. The most frequent reasons why these ideas were excluded are: low chance of R&D success, low benefit potential, close relationship with other candidates, area too broad, area too narrow, and poor idea definition.

### B.1 Technical

#### B.1.1 General

B.1.1.1 Presentation and Manipulation. Human factors and the characteristics of problems being solved or tasks being performed are important in achieving the best interface between machines and software engineers. In addition to issues such as media and perception, there are such aspects as condensations, and/or what one needs to see simultaneously.

B.1.1.2 Rigorous Documentation. Much documentation currently produced and used is neither rigorously machine generated from the software, nor rigorously verified in the terms of formal verification. Automatic generation of documentation for users, maintainers, and operators is one approach. Another is to attach a condition to each documentation fragment such that the condition must still be true or the documentation fragment can no longer be considered valid. Less rigorously, one might attach to a documentation fragment a condition that insists that there be no change in certain parts of the software if the fragment is to continue to be considered valid. Rigorous documentation would not only provide users of the documenta-

tion with confidence in its validity, but could also greatly facilitate documentation maintenance.

**B.1.1.3 Conflict Recognition Among Representations.** Separately prepared descriptions or representations of software could be compared for consistency. This is needed not only for different representations of the same system during one part of the lifecycle, but also across parts of the lifecycle, for example between a representation of the requirements and a representation of the design.

**B.1.1.4 Exploratory Systems Applications of VHSIC.** While standardization of hardware is the correct direction in which to be driving, by the end of the decade VHSIC technology should embody applications in hardware relatively quickly and inexpensively. The placing of frequently used logic in hardware could offer significant speed improvement. The conditions under which this would be desirable and the methods and implications of this placement would be explored.

**B.1.1.5 Military Information Utility.** The concept of a military information utility provides a framework for computer-based information systems for DoD. Information services composed of processing, storage, and communication, along with the relevant input and output devices and media would be thought of not individually, or on an ad hoc basis, but rather in terms of common information utility services to be provided. Different types or classes of service can be provided to different types of users for different purposes. Just as dial-up telephones for tactical communication provide a more flexible and easy-to-use system than a special purpose net, so might an information utility provide more flexible and understandable service than a special purpose information system. The Source and the National Software Works provide embryonic examples. An example of a standard service might be graphic map displays, including standard symbols for objects of interest. The integrated software support

environment might be a class of service under a military information utility.

B.1.1.6 Multiple Classes of Service. What are the classes of information services needed within DoD? Classes of services might be established by analyzing requirements by user, types of functions performed, and performance level required. Service classes for both users and technicians would be needed.

B.1.1.7 Standard Real-Time Operating System. A standard operating system for military computers (or a standard operating system interface) would simplify the conversion of applications between computers. Operating system incompatibilities inhibit software reuse, even when the same programming language is available on all systems. This may be an alternative approach to the standardization of instruction sets for military computers, or both ideas may be useful.

#### B.1.2 Requirements

B.1.2.1 Rapid Derivation of Requirements. Establishing requirements is presently a long process; the goal is a capability to develop requirements quickly after a decision is made to proceed. Methodologies need to be developed or selected. These methodologies may be manual, computer-based, or some combination thereof. It is likely that methodologies will differ in detail by application or by problem domain.

B.1.2.2 Transform Informal to Formal Requirements. Requirements usually originate in an informal expression. Expanding an informal statement into a rigorous and complete set of requirements can be very difficult and expensive. New methodologies, advances in automated question and answer systems, and such approaches as query by example are all possibilities.

B.1.2.3 Requirements Languages Translation. If different requirements languages are used for different problem domains, then it will be useful to be able to translate one language into another. This may not always be straightforward, since some features of requirements languages are not common in normal programming languages. It is possible that one might want to adopt a canonical requirements language and translate all the others into it.

B.1.2.4 Weakest Possible Requirements Description. In order to have the maximum design and implementation freedom, requirements should be no more restrictive than necessary. A theory of requirements and procedure description strength is needed, giving the dimensions along which requirements can be stronger or weaker. Currently there are at least two dimensions being widely discussed: concurrency (freedom to process in parallel) and indeterminacy (freedom to choose any one action from a set).

### B.1.3 Design

B.1.3.1 Derivation of Software from Specifications. Since an application-oriented specification is presumably easier to prepare and validate with the user, it would be useful to be able to turn such a specification quickly into software executable with reasonable efficiency. Automatic derivation simply means another very high level language. In situations where automatic generation is not well understood, a computer/human team might be used.

B.1.3.2 Very High Level Languages. The ideal is to be able to do anything one wants to do with a few natural commands. Very high level languages might look like executable requirements languages. Must very high level languages be tailored to specific application problem domains, or are there generic approaches usable over wide areas? The success of very high level languages depends on their match with users' conceptualizations.

B.1.3.3 Component Tailoring and Interfacing. Building blocks might be used to construct a system through a question and answer session with a tool. The tool would place software components together, tailor them properly, and interface them so that they work together.

B.1.3.4 Publication of Standard Designs. Much software being built today is very similar to other software already built, with similar applications and functions, or similar data and data structures. The publication and dissemination of carefully thought out and verified designs will reduce a significant amount of time now spent re-inventing the wheel.

B.1.3.5 Data Structure and Abstraction. Two concepts widely espoused by researchers are modeling a system's data base structure on actual, external relationships and encapsulating portions of the structure in modules that both regulate access and hide implementation details. The entity-relationship approach to data base design and some knowledge-based systems approaches are examples of the former. Packages in Ada are specifically intended to be used for data abstraction or encapsulation.

#### B.1.4 Programming

B.1.4.1 Code Skeletons. Across broad classes of applications, software is much the same at a high level, and differs only in details. For the invariant high-level, standard code skeletons that can be fleshed out with details allows the quick production of such software. This is related to self-interfacing software.

B.1.4.2 Graph-Oriented Language. There is a lack of well designed, easy-to-use graph-oriented languages. Graphs (nodes and connectors), however, are a natural way to describe many problems, including many problems in computer science.

B.1.4.3 Generating Assertions from Requirements. The essential condition for proper software is that it meet its requirements. The generation of assertions to be placed in code from the requirements facilitates formal verification or dynamic testing to show that requirements are met. The generation of assertions involving decomposition of requirements conditions might also be dependent upon design and not just upon requirements.

B.1.4.4 Transform to Satisfy Physical Constraints. Implementations may be transformed to fit different configurations, or they may be transformed to optimize various resource usages. This is related to configuration independence.

B.1.4.5 Man-Machine Quality Improvement Team. Completely automated software quality improvement may be impossible, particularly when semantics are involved. For example, there may be a need for a person to supply a functional name for a new module produced by automatic restructuring. The machine could supply the discipline and processing power, and the person could supply real world knowledge.

B.1.4.6 Application Generators. Application generators are software tools that generate programs within a limited application area or range of program types. Products and research results range in scope from data entry screen formatters to inventory systems. Application generators have generally been ad hoc approaches in limited areas, but they can provide substantial increases in productivity nevertheless.

B.1.4.7 Reusable Software. If significant software modules could be re-used, fewer new software modules would need to be written. This will significantly increase programmer productivity. Royalties or other incentives could be used to encourage programmers to design, document, and maintain software to this end.

B.1.4.8 Actor Languages. In the performance of distributed tasks, it is often convenient to think of each process as an actor

who performs his role and communicates with other actors via messages. Actors are humans or knowledge-based problem-solving systems. As originally conceived, an actor system is a model of a small scientific community in which each actor is a scientist, but the analogy to military problem-solving is straightforward. Communication is via stylized messages; the purpose is to seek solutions within the community by proposing new plans, refining them, criticizing refinements, and proposing consensual solutions. Smalltalk is an example of an existing actor language.

#### B.1.5 Testing

B.1.5.1 Static Analysis of Software. Many analysis and verification techniques can be applied to software without execution. Among these are: formal verification, data flow analysis, standards and style auditors, and interface checkers. The development of comprehensive but expandable tools that are both easy-to-use and reasonably efficient would simplify the use of these techniques.

B.1.5.2 Generating Test Data from Requirements. Requirements are statements of what software must do. If test data could be systematically developed from requirements statements, then this data could be used to verify compliance. A complicating factor is testing to ensure that the system not only does what it should do, but also that it doesn't do what it shouldn't do.

B.1.5.3 Generating Test Data to Violate Assertions. Given that assertions have been placed throughout the software, a test data generator can use search techniques that attempt to violate more and more assertions. The assertions, in essence, provide an automated way of checking expected results.

B.1.5.4 Testbed Facilities. Modular, easily changed testbed facilities could be provided to facilitate testing various classes of

embedded systems. Each platform type or class of embedded system might require its own facility, although many components could likely be shared. Testbeds would offer variable levels of simulated and real environments.

#### B.1.6 Operations

B.1.6.1 Construction for Future Evolution. How should software be originally designed and built so that it can survive evolving requirements? Modularity and pre-planned product improvement are examples of approaches.

B.1.6.2 Modification of Large Systems. Changes in requirements need to be implemented, and repairs need to be made. How this can be done quickly without introducing error or requiring complete reverification is a problem.

#### B.2 Managerial

##### B.2.1 General

B.2.1.1 Model Contracts for Buying Software. Model contracts for acquiring software would be very useful if they contained the proper technical and managerial concerns. Among things to be included are: software metrics, considerations for software personnel quality, standards and practices, software prototyping, verification and validation, and proper contract incentives. How construction should be broken up and what types of contracts should be used for each stage of the lifecycle would also be covered.

B.2.1.2 Maximizing DoD Rights to Software. Rights to software and particularly software tools can be problems. Proper contracting and acquisition management are needed to avoid excessive costs

incurred because the original contract did not provide for delivery, ownership, or full Government rights to tools, support software, and documentation (sometimes considered proprietary) throughout the life-cycle, particularly the later (maintenance/operation) stages.

B.2.1.3 Multiplying Expert Effectiveness. Off-loading some activities from scarce and expensive highly skilled software personnel to less expensive ancillary personnel could increase effectiveness and decrease costs. The chief programmer team concept is one attempt in this direction.

#### B.2.2 Conception/Feasibility

B.2.2.1 Quick Look Feasibility/Evaluation. It has been estimated that, on the average, more than five years pass between the time a DoD system is conceived and the time it begins a formal requirements analysis. Better abilities to analyze the feasibility of potential systems quickly and to evaluate alternatives so that an approach can be chosen might put systems in the field years earlier than is now the case. Approaches that might help in this include risk management of evolving systems and simulation of proposed systems. This is a management as well as a technical problem that will probably require a managerial as well as a technical solution.

#### B.3 Continuity-Related

##### B.3.1 General

B.3.1.1 Completely Captured Software. If everything needed to understand a system were recorded, then learning about the system, analyzing it, maintaining it, and changing personnel would be easier. Information could be captured concerning alternatives considered,

design choices made, and unwritten assumptions made regarding future modifications.

B.3.1.2 Multi-person Machine Mediated Programming. Team programming is becoming widely accepted as the best approach. It is sometimes difficult, however, to bring all persons whom one would like to have on the team together, because of geographic separation or scheduling conflicts. With a computer and associated communications as the interface among team members, geographic and time schedule problems could be reduced. Also, additional discipline and organization could be provided by the computer. The National Software Works points in this direction. A bonus from having all conversations processed through the machine is that they can be captured to provide potentially valuable documentation concerning the background and rationale for the implementation.

B.3.1.3 Totally Visible Software. The intangible nature of software has led to a number of managerial, user, and buyer misunderstandings. If all relevant data concerning a software system could be captured, then the data could be analyzed and displayed in terms that managers and users could understand. Comparison to plans and measures of progress and quality will be needed as well as means to learn what software exists.

B.3.1.4 Systems that Never Forget. With the advent of less expensive storage (for example laser-video disk), it becomes attractive to consider a strategy of always being able to return to some previous state. Confidence and security from disaster are added benefits. The integrated software support environment might particularly benefit from such an approach. Tools would be required to locate and retrieve information of interest easily, in order to derive full benefits.

### C. SOFTWARE PROBLEM AREAS

This appendix presents a comprehensive outline of software problem areas. It is the result of repeated efforts to create a structure containing and classifying all problem areas found in readily available prior studies. (Those studies are reviewed in Appendix D.)

The outline contains four primary categories: (1) Technical, (2) Managerial, (3) Personnel-Related, and (4) Continuity-Related. Problem areas mentioned in previous studies are inserted in the structures at appropriate points.

- o Technical includes all steps that go into producing quality results at each phase in the software lifecycle. In addition to exact, measurable definitions of excellence and utility, this category includes procedures and tools to help produce excellent, useful work. This primary category includes the factors of Excellence, Utility, Efficiency, and Quality Control. Briefly, Excellence includes the accurate transformation of inputs to a phase into outputs from a phase and assurances that such transformations are correct, complete, internally consistent, and dependable. Utility includes ease of use, clarity, relevance to the problem, and lack of unneeded complexity. Efficiency ensures that the development, operation, and maintenance processes, and the finished product are sparing of resources (time, money, machines, and people); that the implementation is clean, without wasted motion or needlessly complex operations; that appropriate risks are taken and that they succeed; and that the final product functions efficiently. Quality Control ensures that the finished product of each stage meets established standards within acceptable tolerances.
- o Managerial includes methods for collecting information on project progress and resource utilization, and for predicting future progress in terms of time, money, personnel, and other resources, despite incomplete and incorrect data. It also includes methodologies for exercising management control over budgets and schedules.
- o Personnel-Related is a category because it is usually the

scarcest and most critical resource in the software life-cycle, and has a direct effect on cost. This category includes finding, training, and retaining good people.

- o Continuity-Related includes all written and oral communication involved in a software project: requirements statements, design documents, specifications, manuals, memoranda, and the project history, as well as oral communications among members of the software team and between team members and users. It is involved with the accurate, complete, and internally consistent transmission of information. It includes methodologies and tools to ensure that information can be traced throughout all stages of the software life-cycle.

Many problems were analyzed to arrive at 19 major problem areas. Almost all problem areas are common to all application areas and types of programming. Problem areas are summarized in Figure 7 and are discussed in detail in the following subsections. References in the form [source:pages] are given after most paragraphs; the bibliography to which the references refer is at the end of this appendix.

### C.1 Technical

Technical has eight primary problem areas, as shown in Figure 7. (1) flawed and conflicting standards, (2) inappropriate constraints, (3) poor definition of goals and measures (e.g. incorrect, unusable success criteria), (4) faulty design, (5) incorrect selection and use of languages and packaged software, (6) poor use of implementation tools, (7) inferior testing methodology, and (8) unsatisfactory product evaluation and follow-up.

#### C.1.1 Flawed and Conflicting Standards

As the Software Acquisition and Development Working Group found, "Perhaps the most prevalent difficulty with the software acquisition and development process seen by the software industry representatives is the lack of a consistent software standard not only among the various government agencies, but also often within a single

#### TECHNICAL

flawed and conflicting standards  
inappropriate external constraints  
poor definition of goals and measures; incorrect, unusable success criteria  
    inadequate assessment of goals  
    unrealistic performance assumptions  
    poor priority ranking of goals  
    inappropriately constrained design options  
    inadequate requirements and quality measures for:  
        - user-visible performance  
        - efficiency  
        - error tolerance  
        - reliability  
        - availability  
        - testability  
        - maintainability  
        - support tools  
        - evolution / adaptability  
        - portability  
        - usability  
human factors not considered in user interface  
human factors not considered in internal design  
poor quality measures for products of each development stage  
inadequate architectural analysis  
faulty design development  
    lack of insight into design and testing alternatives  
    undependable predictors for cost vs quality vs time  
    insufficient use of risk reduction tools (e.g. prototypers, simulators)  
incorrect selection and use of languages & packages  
poor use of implementation tools  
    unavailable or unknown programming support tools  
    hard to use programming support tools  
    inefficient programming support tools  
    unevaluated programming support tools  
    nonstandardized programming support tools  
inferior testing methodology  
    lack of dependable, validated verification methods  
    no accepted method for predicting latent bugs and their impact  
unsatisfactory product evaluation and follow-up  
    weak lessons learned feedback loop

#### MANAGERIAL

weak project leadership and coordination  
    poor management planning  
    ambiguous policy guidance  
    incorrect placement of resources  
    inadequate configuration and change control

Figure 7: Problem Areas

MANAGERIAL Cont'd

- unclear lines of responsibility and authority
- non-uniform enforcement of rules
- insufficient discipline and rigor
- undue influence of external factors in scheduling/budgeting
- poor monitoring & prediction of schedules & budgets
  - inadequate baselines for project scheduling and budgeting
  - unsatisfactory tools for predicting schedules and budgets
  - lack of accurate models for sizing tasks
  - no general agreement on the data to be used to indicate project status
  - poor selection and gathering of data on current project status
  - insufficient risk analysis
- unsatisfactory project control
  - schedule slippage and budget overruns
  - difficulties with standard life cycle model
- flawed methodology for the acquisition process
  - incorrect contract type and contractual requirements
  - patchy contract monitoring
  - clumsy contract control

PERSONNEL-RELATED

- inability to attract and keep qualified personnel
  - high turnover
  - shortage of qualified personnel
  - substitution of unqualified personnel
  - understaffed projects
  - incorrect job classifications and pay scales
  - reversed work incentives
  - "dead end" career ladders
  - clearance problems
- unsuitable selection criteria and measures of competence for personnel
- poor exploitation of personnel
  - poor training
  - slow technology transfer

CONTINUITY-RELATED

- ambiguous, unclear, incomplete communications
  - invalid assumptions
  - inconsistent standards and terminology
  - lack of structure in communications
  - insufficient review
- slow, outdated communications
  - obsolete documentation
- lack of project history
  - loss of tradeoff study results and reasons for design decisions
- poor phase-to-phase continuity
  - forgotten requirements and specifications

Figure 7: Problem Areas (concluded)

agency."[10:p. 2-1] Terms used in standards are not themselves standardized, the required level of detail is unclear, and the standards are imprecise. The multiplicity of standards and of interpretations of standards forces workers to spend time studying variations on standards instead of producing. In addition, standards are not consistently enforced; they often result in generation of paper that is never read. Good standards are missing in critical areas such as contract management and software quality. Worse yet, standards that are present are so poor that they often hinder, rather than help. [See 10: pp. 2-1 thru 2-3, 3-2; 5: pp. I-7 thru I-8] [11: 2-17]

#### C.1.2 Inappropriate Constraints

A prime example of an inappropriate constraint is the use of government-specified hardware. (Other such inappropriate constraints occur when the contractor is required to use unsuitable on-site personnel, or an unsuitable design or design methodology.) Hardware costs are falling and software costs are rising, yet the government may constrain the contractor or system developer to use inappropriate hardware solely because it is available or inexpensive. Difficulties and expense caused by attempts to force software into such hardware are disregarded. [See 10: pp. 2-20 thru 2-21, 3-4]

#### C.1.3 Poor Definition of Goals and Measures

Poor definition of goals and measures of success may well be the most important problem. Initial goals are often too ambitious: a complex system is specified to be built in one massive effort instead of stages. Goals for the performance and capabilities of the finished system are often not validated or subjected to a thorough cost/benefit and risk analysis before being incorporated into requirements and specifications. Because these analyses were not performed, ranking of goals by priority is often not done or is incorrect. Sometimes, needed goals are omitted entirely, because the person setting goals has made an unstated assumption; e.g. that the

system to be delivered is a standard product not requiring goals to be set for its performance. Such assumptions can easily lead to disaster, because there is no way of measuring the quality of the delivered product if there is no comprehensive set of goals for it to meet.

The process of setting goals is also plagued by inappropriate levels of detail, in which a specific solution is written into requirements instead of into a statement of the problem to be solved, thereby constraining the designer unnecessarily. As was stated in Stockton Gaines's "Draft Plan for the DoD Software Technology Initiative,"

"There is substantial evidence that requirements rarely capture the desired intent, and even less often in the best form. As a consequence, specifications often both over-specify certain aspects of the system and under-specify others. Specifications frequently specify minor details while omitting major aspects that should have been specified. The relationship of specifications to evolution and design, in the context of software, needs special attention." [7: p. 2] [See 10: pp. 2-3 thru 2-5; 11: p. 2-3; 1: pp. 2-7, 2-8, 3-5; 8: p. 4]

Even when goals are stated, they usually fail to meet standards of excellence. There is usually a severe lack of usable, comprehensive, measurable, validated requirements for user-visible performance (e.g. transactions per minute), efficiency, error tolerance, reliability, availability, testability, maintainability, support tools, evolution-adaptability, portability, and usability. In addition, there is often a lack of consideration of human factors when implementations and maintenance tools are considered. Just how large should a module be if it is to be comprehended as a whole? And how should the user interface for a maintenance tool be designed, if it is to be easy to use and to help reduce errors? The complexity of designs often increases beyond the level of human comprehension.

[See 1: pp. 2-11, 3-6, 3-34; 11: p. 2-9; 12: p. 35; 5: pp. 1-2, 1-3, and 1-12; 4: p. 39; 6: p. 8; 8: p. 4]

All the above problems contribute to a key problem in the area of goal definition: the lack of dependable, usable measures of quality for evaluating concepts, requirements, specifications, implementations, test plans, and finished products. Without understandable and acceptable measures, goals cannot be rigorously defined, and without rigorously defined goals, excellence cannot be procured or consistently rewarded.

#### C.1.4 Faulty Design

Problems in this area often start with the choice of an incorrect system architecture, based on inadequate analyses of alternatives and trade-offs. The reason for such inadequate analyses is the lack of deep insight into design and testing alternatives and the resulting lack of dependable predictors for cost vs. quality vs. time trade-offs. Therefore, it is not surprising that reasonable automated tools are not in use. Indeed, there is a lack of good development and evaluation tools, and a lack of good risk reduction tools such as simulators and prototype construction systems. Lack of good tools for design development also harms system evolution, because the impact of proposed modifications cannot easily be determined. With today's multi-year system development cycles, evolutionary developments are virtually inevitable, if only to allow designers to catch up with changes in the environment between system specification and initial operation. [See 1: pp. 3-8, 3-10; 2: pp. 12, 13, 25; 4: pp. 23, 32, 33; 10: pp. 2-7, 2-8]

Poor implementation is the production of overly complex, unclear, error-prone software in an inefficient way. Poor use of languages, packaged software, and support tools reduce programmer productivity.

#### C.1.5 Incorrect Selection and Use of Languages & Packages

An inappropriate language may be selected for an application, while existing software is ignored, although it could be modified. Languages proliferate; therefore, time is needed to retrain programmers. Even standard languages may be implemented differently on different machines. When programmers become accustomed to using one manufacturer's language extensions, their programs lose portability. Since many currently-used languages do not encourage clean, modular, structured design, non-portable code often cannot easily be modified to make it run on a different system. [See 14: pp. 176-177; 11: p. 7-2; 5: p. 1-4; 4: pp. 35 thru 41]

#### C.1.6 Poor Use of Implementation Tools

Software tools (e.g. editors, debuggers, library maintainers) are often unavailable, unpublicized, hard to use, inefficient, and unevaluated. Implementers are often unaware of tools or previous work (subroutines, etc.) that could save time. In addition, lack of tool and language standardization makes time lost to retraining an important factor whenever a new tool or language is considered. Lack of tool portability raises the probability that such retraining will be necessary whenever a programmer is moved from one project to another. Sometimes a tool (such as a specialized cross-compiler) will work only on special hardware or on a costly machine, and often the tool or machine is not available to the Government for maintenance after system acceptance. The cost of using a tool, including tool acquisition and user training, must be weighed against the benefit obtained by tool use. Unfortunately, there are few studies providing the necessary comparison data. [See 14: pp. 173, 174; 11: pp. 2-11, 5-21; 4: pp. 35, 36; 10: p. 1-6]

#### C.1.7 Inferior Testing Methodology

There is no satisfactory methodology to determine how much testing is enough; there are few dependable verification methods (and

methodologies for choosing the appropriate method), and there is no dependable methodology for predicting the number of latent bugs in a system or their impact on availability and reliability. Not enough money is allocated for testing. Testing plans are validated too late, if at all; and the testing phase is far too time-consuming. Test drivers often have their own errors, as do test data sets, and test results are inconclusive. It is difficult to make the test environment an accurate representation of the operational environment. [See 14: pp. 47, 59, 60; 11: p. 2-19; 5: pp. 1-5, 1-12, 1-13; 10: pp. 2-8, 2-14]

#### C.1.8 Unsatisfactory Product Evaluation and Follow-up

Quality Assurance is poor in the software industry, because success criteria are often inappropriate, incorrect, and unusable. The lack of a "lessons learned" feedback loop to the software development community is a serious omission, as it is only through the use of such a mechanism that data needed for research on goals and measures of success can be accumulated. [See 1: pp. 11 thru 14; 5: p. 1-5; 10: p. A5]

### C.2 Managerial

There are four major problem areas in management, as shown in Figure 7: (1) weak project leadership and coordination, (2) poor monitoring and prediction of schedules and budgets, (3) unsatisfactory project control, and (4) flawed acquisition methodology.

#### C.2.1 Weak Project Leadership and Coordination

All too often there is a lack of a good system development plan, symptomatic of a general lack of policy guidance and planning. A standard method for selecting a management methodology is lacking. Emphasis and resources can be placed on the wrong areas, and subtasks may be poorly synchronized and interfaced. Change control is not rigorously managed, and coordination of various program segments

rapidly breaks down. [See 11: p. 2-7; 13: p. 53; 10: p. 1-4; 14: p. 68]

A clear assignment of responsibility and authority is often missing, for both individuals and organizations. Rules are sometimes promulgated without data showing that they are valid or that compliance is measurable. Because rules are not enforced uniformly, there is a general lack of discipline and rigor in all phases of the life-cycle, which is exacerbated by management's reluctance to discipline an indispensable programmer, who might quit. [See 11: p. 2-12; 1: pp. 2-2, 3-19; 9: pp. A-31, A-37; 6: p. 8; 14: p. 68]

The tendency to align software checkpoints or mileposts with external events (e.g. hardware and manpower availability, political considerations), regardless of software development realities, causes more trouble. Such alignments sometimes result in premature programming, before the design is finished, and in concurrent development of hardware and software. [See 11: p. 5-6; 1: pp. 3-20, 3-21]

#### C.2.2 Poor Monitoring & Prediction of Schedules & Budgets

This problem has a number of contributing factors. One is the severe lack of meaningful baseline data about schedule and budget experiences of previous projects. The Rome Air Development Center is currently gathering data, but existing data are sparse and of poor quality. Data must be based on, or convertible to, standard metrics and contain information on project type, development strategy, etc. There are no generally accepted models for predicting schedules and budgets and for sizing tasks. The lack of good models makes resource allocation and trade-off analysis difficult. Unrealistic cost and schedule estimates are often the result, because they are based on inadequate risk analyses and unfounded assumptions in the absence of usable models and data. Mission requirement uncertainties, probabilities of changes in system requirements, risks in undertaking a software development at the state of the art, hardware and interface

dependencies, etc. are often not quantified because usable, standard methods simply do not exist. [See 11: p. 7-2; 1: pp. 2-6, 2-8, 3-10, 3-36; 5: p. 1-2; 12: pp. 35, 86, 88; 4: pp. 14 thru 17; 10: pp. 1-4, 1-5, 2-13, 2-16 thru 2-20, 14: p. 54; 8:]

Even with generally accepted, accurate models, the lack of adequate data on project status would continue to create problems. Inadequate cost and schedule monitoring is common, due in large part to poor software progress visibility: the ability to see the percentage of a given job that is satisfactorily completed at any given time. Unfortunately, there is no general agreement on what data accurately reflect the status of a project; therefore, no standardized measure of how close a project is to completion. [See 11: pp. 2-4, 2-6; 1: pp. 2-8, 3-19; 10: p. 2-9]

#### C.2.3 Unsatisfactory Project Control

Schedule slippages and budget overruns are distressingly common, but unsurprising in light of the problems mentioned above. The traditional phasing of a software project, with formal reviews and structured procedures taken from hardware production, can cause various problems (non-productive paperwork, inappropriate rigidities, etc.); however, there are no acceptable alternatives. Management is often uninformed on project status, and unable to affect it directly. Problems in synchronizing hardware production with software production are common. [See 11: p. 5-6; 4: pp. 15, 16; 10: pp. 1-4, 1-6]

#### C.2.4 Flawed Methodology for the Acquisition Process

There is a lack of a standard methodology for handling contracts, and the contract type is sometimes inappropriate for the problem (e.g. fixed-price for a high-risk, poorly-specified development), resulting in loss of control over the contractor or in poor work. Sometimes, software is not isolated from a systems contract to ensure that all rights to it will be acquired. Often, tools used to develop and maintain software are not acquired along with the

software, or money is not provided for their full development, documentation, and delivery. Too few risk reduction contracts (e.g. parallel development, prototyping) are let, and requirements do not stabilize early enough to be handled easily by standard methods. [See 11: p. 2-17; 1: pp. 2-9, 2-10, 3-6, 3-20ff; 5: pp. 1-7 thru 1-9; 2: p. 2-19; 10: pp. 1-4, 1-5, 2-21 thru 2-23, 3-1ff.]

Problems often occur in managing interfaces among software and hardware controlled by different contractors, because of weak configuration control. Sometimes the acquisition manager will become too involved in the internals of software components, managing trivial details from the highest levels of authority. [See 11: p. 7-2; 1: p. 3-7; 5: p. 1-2; 8: p. 4]

Documentation required and delivered during the life of the contract is often no great help in understanding project status. Document requirements can be overlapping and vague, with the result that documentation is overwhelming, hard to understand, and strangely or illogically organized. [See 2: pp. 2-11, 2-40]

The time-consuming procurement process and requirements changes during the life of the procurement cause considerable trouble. Another problem caused by lengthy procurement is staff turnover, both on the side of the contract monitor and on the side of the contractor. Project control can be inconsistent; the project can easily lose or reverse direction at each change. [See 9: p. A-30; 4: p. 17]

### C.3 Personnel-Related

There are three major problem areas under this heading: (1) severe shortage of qualified personnel and difficulties in retaining them, (2) unsuitable selection criteria and measures of competence for personnel, and (3) poor exploitation of personnel.

### C.3.1 Problems Finding and Keeping Qualified Personnel

There is a severe shortage of qualified personnel and a tendency to substitute large numbers of unqualified or poorly qualified workers when qualified workers cannot be found. Not only is the work of poor quality, but the amount of time available for productive work per worker decreases because of the increased time spent in meetings and communications among members of the workforce. In addition, projects often operate with too few seasoned professionals during critical early stages. One reason for the shortage of workers is a national indifference to early education in computer applications and related technologies. Furthermore, there is insufficient emphasis on software engineering in computer science education programs. Other important reasons for the personnel shortage within the Defense community are the overclassification of programming projects, the shortage of capable workers having the required security clearances, and the difficulty in transferring clearances and billets when required. [See 11: pp. 2-15, 5-6, 7-1; 5: p. 1-10; 10: pp. 1-5, 2-23, 2-24, 3-5; 3: p. 13]

Partially due to the severe personnel shortage, there is difficulty in attracting and retaining qualified personnel. Staff turnover is astounding. One estimate is that the typical staffer remains with one job for less than 24 months, despite the fact that the typical software development cycle lasts more than 40 months. [See 11: p. 2-14; 1: p. 2-9; 5: p. 1-8; 4: p. 20]

There appear to be a number of reasons for this situation. Inability or unwillingness to pay salaries commanded by top people is a contributing factor, as is the poor quality of leadership found on many projects. Work incentives are often reversed: competent personnel are overworked, but inadequately challenged, while their pay may be unrelated to competence. (That is not too surprising; there is no generally accepted, objective measure of competence.) Career path

problems contribute to the confusion; excellent technical people are often promoted out of their fields into management, despite the fact that their expertise is needed at the worker level. Indeed, if they are not promoted, technical staff members will often quit or become frustrated. Government job classification and civil service systems can cause quite a bit of trouble: job descriptions are often misleading; job grades are often incorrect and inconsistent from one department to the next; and the tenure system for raises and layoffs often discriminates against the best workers in favor of those longer employed. Since career ladders for computer personnel may end somewhere in middle management, the ambitious staffer must leave the computer field to succeed. Computer specialists in high-security areas have the additional worry that they will become pigeonholed because of their clearances; because there is such a severe shortage of competent professionals with high clearances, they may not be allowed to transfer. [See 11: p. 2-14; 5: p. 1-10; 9: pp. 14, A-16, A-46; 10: pp. A-8, A-9; 8: p. 4]

#### C.3.2 Unsuitable Competence Measures

Little work has been done to define the personality types that would make the best analysts, programmers, programming managers, etc.; for that matter, it is difficult to define "best." The personality type most common today among programmers and analysts may not be the best for the future, but may simply be an artifact of current selection methods, training, and work environment. Because there are no generally accepted measures of excellence in specification, design, testing, working with users or management, etc., there is no way to judge whether the currently most common personality type would be the most successful in achieving excellence across the board. As was stated in [1: p. 53], "There is little organized knowledge of what a software designer does, how he does it, or of what makes a good software designer."

AD-A102 180

MITRE CORP MCLEAN VA

F/G 5/1

CANDIDATE R&D THRUSTS FOR THE SOFTWARE TECHNOLOGY INITIATIVE. (U)

MAY 81 S T REDWINE, E D SIEGEL, G R BERGLASS F19628-81-C-0001

MTR-81W00160

NL

UNCLASSIFIED

3 of 3  
4-81  
4-81-100

END  
DATE  
FILMED  
8-81  
DTIC

### C.3.3 Poor Exploitation of Personnel

Poor training is a major problem for available personnel. Not only do workers lack important knowledge, but managers lack the technical knowledge to enable them to understand their subordinates and the managerial knowledge to manage them. The slow rate of information transfer from the research institutes to the field is extremely damaging in an area that changes as rapidly as software engineering. A formal, fast, and effective method for technology transfer is needed. [See 11: pp. 2-14, 7-1; 5: pp. 1-8, 1-10; 9: p. A-9; 6: p. 8; 4: p. 20; 10: p. 1-4]

### C.4 Continuity-Related

Figure 7 shows four major problem areas.

#### C.4.1 Ambiguous, Unclear, Incomplete Communication

These problems lead to invalid assumptions on the part of both sender and receiver. Inadequate documentation is often a major cause, as are inconsistent standards and terminology. The sender and the receiver of a message may interpret the same statement in different ways, because they have different frames of reference or different experiences. The team designing the computer system has experience in systems design, but probably doesn't know the application as well as the user. "One cannot expect the contractor's system analysts to be good artillerymen." [2: p. 2-7] It is important that unstated assumptions on the user's part be expressed and made clear to implementers, and vice versa. A lack of structure or a poor structure in communication often results in undetected omissions or internal contradictions leading to system changes as clarifications are made. Insufficient review, combined with the problems in reviewing an unclear, poorly structured document, result in the common situation of incomplete and ambiguous requirement statements, specifications, user documentation, maintenance records, etc. [See 11: pp.

5-2, 5-6; 2: p. 2-7; 5: p. 1-2; 6: pp. 8, 9; 10: pp. 1-4, 2-1 thru 2-5, A-3]

#### C.4.2 Slow, Outdated Communications

Large numbers of changes are especially dangerous, because of communications that are out of date. Obsolete documentation is typical; the slow dissemination of up-to-date information often results in systems that are obsolescent before delivery. [See 10: p. 2-3]

#### C.4.3 Lack of a Project History

Again, inadequate, incomplete documentation is a typical example, along with failure to obtain support software and maintenance documentation. Without an adequate project history, a project will find it quite difficult to recover when a critical staff member leaves. Failure to document design decisions and the analyses that led to them can also waste a large amount of effort if those decisions are ever reopened because of a change or reevaluation of the system. [See 11: p. 2-5, 1: p. 3-15; 10: p. 1-4]

#### C.4.4 Poor Phase-to-Phase Continuity

The final problem is that of poor continuity from one software phase to the next: disparity between requirements and specifications, between specifications and code, between requirements and test plans, etc. Some work is being done to provide better continuity, but current practice is still generally poor. No standard method exists to trace all system requirements through all software development phases, or to ensure that all design information is accounted for and handled in each relevant phase. [See 13: p. 53]

#### C.5 Bibliography

[1] Asch, A., D. W. Kelliher, J. P. Locher III, and T. Connors. DoD Weapon System Software Acquisition and Management Study. MITRE

Corp., Bedford, MA, tech. report MTR-6908, May 1975, vol. 1, MITRE Findings and Recommendations.

[2] Asch, A., D. W. Kelliher, J. P. Locher III, and T. Connors. DoD Weapon System Software Acquisition and Management Study. MITRE Corp., Bedford, MA, tech. report MTR-6908, June 1975, vol. 2, Supporting Material.

[3] Blumenthal, M. "Professor Bemoans Indifference to DP Education." Computerworld vol. 15, no. 16 (Apr. 20, 1981), p. 13.

[4] Defense Computer Resources Technology Plan. Management Steering Committee for Embedded Computer Resources, USD (R&E), Washington, DC, June 1979.

[5] DeRoze, B. C. Defense System Software Management Plan. DoD Software Management Steering Committee, OSD (I&L), Washington, DC, Mar. 1976. (NTIS accession no. AD-A022 558.)

[6] Drezner, S. M., H. Shulman, and W. H. Ware. The Computer Resource Management Study: Executive Summary. Rand Corp., Santa Monica, CA, report R-1855-PR, Sept. 1975. (NTIS accession no. AD-A018 884.)

[7] Gaines, S. "Draft plan for discussion at May 30 Software Tech Initiative meeting." ARPANET mail from RAND-UNIX, 28 May 1980.

[8] Giese, C. "Final Report, Mission Area Orientation, DoD Software Technology Workshop, Ft. Belvoir, VA, 15, 16 May 1980." Letter to Dr. D. Fisher, Director, Electronics and Physical Sciences, OUSD (R&E), Washington, DC, 7 July 1980.

[9] Jensen, A. P., E. L. Dreeman, and B. A. Reardon. Information Technology and Governmental Reorganization: Summary of the Federal Data Processing Reorganization Project. Office of Management and Budget, Washington, DC, Apr. 1979. (NTIS accession no. PB-294 543.)

[10] Jones, V. E., chairman. Final Report of the Software Acquisition and Development Working Group. ASD (C3I), Washington, DC, July 1980.

[11] Kossiakoff, et al. DoD Weapon Systems Software Management Study. Johns Hopkins Applied Physics Laboratory, Laurel, MD, report APL/JHU SR 75-3, June 1975. (NTIS accession no. AD-A022 160.)

[12] Kosy, D. W. Air Force Command and Control Information Processing in the 1980s: Trends in Software Technology. Rand Corp., Santa Monica, CA, report R-1012-PR, June 1974.

[13] Thayer, R. H., A. Pyster, R. C. Wood. "The Challenge of Software Engineering Project Management." Computer vol. 13, no. 8 (Aug. 1980), pp. 50-59.

[14] Wegner, P., ed. Research Directions in Software Technology. Cambridge, MA: MIT Press, 1979.

#### D. SUMMARY OF SOME REVIEWED STUDIES

Outlines of some previous studies are shown below, giving the funding, objectives, methodology, and structure of each study.

Research Directions in Software Technology, Peter Wegner, ed., MIT Press, Cambridge, MA, 1979.

Funded by: ONR, ARO, AFOSR

Objectives: To "set the stage for a research attack on the defined problems by preparing a state-of-the-art summary of...all known approaches which might contribute to the eventual improvement of computer software." (p. xi) Does not contain guidelines for future research.

Methodology: 90 individuals contributed papers that were edited through five drafts. Editorial conferences produced discussion sections. Study lasted from 1975 through 1978.

Structure: Part I -- The Software Problem; Part II -- Research Directions

DoD Weapon Systems Software Management Study, A. Kossiakoff et al., Johns Hopkins Applied Physics Lab, Laurel, MD, June 1975.

Funded by: OSD; DoD Software Management Steering Committee

Objectives: "To identify and define (1) the nature of the critical software problems facing the DoD; (2) the principal factors contributing to the problems, (3) the high payoff areas and alternatives available, and (4) the management instruments and policies that are needed to define and bound the functions, responsibilities and mission areas of weapon systems software management." (p.1)

Methodology: "Review and analysis of ten recent major DoD-sponsored studies....Review of the software design and management in ten Navy and two Army weapon systems...Discussions with service and industry organizations involved in weapon system software acquisition, development, and maintenance." Study lasted from January through June 1975.

Structure: Brief survey of problem areas is followed by summary pages of recommended actions and detailed reviews as described above.

DoD Weapon Systems Software Acquisition and Management Study, Vols. 1 and 2, A. Asch et al., The MITRE Corp. MTR-6908, May 1975.

Funded by: (Same as immediately above.)

Objectives: (Same as immediately above.)

Methodology: (Same as immediately above, with the exception that the reviews were of five Army, nine Air Force, and one joint project.)

Structure: (Same as immediately above, but in two volumes.)

Defense System Software Management Plan, Barry C. De Roze, ASD (I&L), Washington D.C., March 1976.

Funded by: DoD internal

Objectives: To document a comprehensive plan for providing solutions to some of the key problems in DoD software acquisition and management.

Methodology: N/A

Structure: Part I -- Policy, Practice, Procedure, and Technology Elements, indicating problems addressed and actions to be taken; Part II -- Implementation Brief, describing organizational roles, responsibilities, and interactions.

Air Force Command and Control Information Processing in the 1980's:

Trends in Software Technology, Donald W. Kosy, The Rand Corp. R-1012-PR, Santa Monica, CA, June 1974.

Funded by: AFSC

Objectives: "To design an integrated Air Force R & D program for the present decade that would develop the information-processing technology needed to meet probable Air Force command and control requirements in the following decade." (p.iii)

Methodology: Revision and updating of the CCIP-85 report; conservative extrapolation from historical patterns.

Structure: Brief overview of software technology followed by a section on future requirements for Air Force Command-Control systems, sections on the past evolution of software technology and future technology trends, and a conclusions and recommendations section.

The Computer Resource Management Study: Executive Summary, Stephen M. Drezner et al., The Rand Corp. R-1855-PR, Santa Monica, CA, September 1975.

Funded by: USAF

Objectives: To produce recommendations that will help the Air Force manage its computer resources.

Methodology: Visits to Air Force projects that contain "a substantial computer component."

Structure: A statement of the problem followed by observations and recommendations.

Federal Data Processing Reorganization Study, Office of Management and Budget, Washington, D.C., 1978-1979.

Funded by: OMB internal

Objectives: "To examine the ways in which the Federal Government acquires, manages, and uses data processing technology and to make recommendations that will help the Government (1) improve the delivery of services through the effective application of computer and related telecommunications technology; (2) improve the application and management of the relevant resources; (3) eliminate duplication and overlap in agency jurisdiction relative to computer issues; and (4) improve the productivity of the Federal data processing work force." (p.1)

Methodology: Fifty-five computer professionals from the public and private sectors divided into ten independent study teams to perform reviews of all relevant documents and to conduct an

extensive series of interviews.

Structure: Recommendations from study, followed by reports from the ten teams in separate reports.

Defense Computer Resources Technology Plan, Management Steering Committee for Embedded Computer Resources, USD(R&E), Washington, D.C., June 1979.

Funded by: DoD internal

Objectives: "To provide coherent direction and guidance to generic computer technology R&D efforts for a period of at least five years (FY1980 - FY1984)." (p. 5)

Methodology: Develop plans and recommendations made by study groups in the 1974-1976 time frame; refine plans with the assistance of the three Services as well as of DARPA, DCA, and NSA.

Structure: A statement of objectives and program management is followed by summaries of the technology areas (Life Cycle Management Tools, System Design and Architecture, Software Product Specification and Standardization, Computer Hardware). Each summary presents problem areas, technical issues and approach, and research direction and action.

Final Report of the Software Acquisition and Development Working Group, Victor E. Jones, chair, ASD(C31), Washington, D.C., July 1980.

Funded by: DoD

Objectives: "Determining the efficacy and cost effectiveness of current software acquisition and development practices within the Intelligence community, and ascertaining areas which could benefit from better management controls." (p. ii)

Methodology: Software development corporations doing business with the Intelligence Community were invited to present their views, and case histories were studied.

Structure: Recommendations are followed by summaries of the industry comments and the case histories are broken down into categories (e.g., Documentation Standards, Software Development Management).

Computer Technology Forecast and Weapon Systems Impact Study (COMTEC-2000), COMTEC-2000 Study Group, HQ AFSC TR 78-03 vol. 1 - 3, Washington, D.C., December 1978 - July 1979.

Funded by: Air Force Systems Command

Objectives: "Forecast the advancement of computer and computer-related telecommunication technologies; assess the potential impact of these technology advances on the capabilities of existing or future Air Force systems through the year 2000; determine the policies and R&D initiatives required to bring these technology advances to fruition and to incorporate them in future weapon system capabilities." (vol. 1, p. 1-1)

Methodology: The technology forecasts were prepared by government and industry specialists during a week at the Air Force Academy, and the system impact studies were performed the following week by Air Force and industrial weapon system planners

and developers. The COMTEC-2000 Steering Committee then analyzed the results of the meetings and synthesized twelve major issues that were studied by small working groups.

Structure: The three phases of the study are issued in three volumes: Volume 1 summarizes the first two phases, volume 2 provides technical data for those phases, and volume 3 presents the results of the third phase.

Army Software Technology R&D Program, Technology Transfer and Organization Plan, Software Technology Division, CENTACS, CORADCOM, Fort Monmouth, NJ, Sept. 1980.

Funded by: Army

Objectives: "to provide the requirements for and definition of the Software Technology Division as an organizational component of CENTACS, CORADCOM, to identify, implement, and introduce into operations software development tooling and other products in response to the urgent need detailed in [Post Deployment Software Support (PDSS) Concept Plan Battlefield Automated Systems, May 1980]." (p. 1)

Methodology: An organization and set of tasks was proposed after a study of the problem areas mentioned in the PDSS study.

Structure: A review of the PDSS study is followed by a proposed structure for the Division and a set of task descriptions.

Preliminary Master Plan for Tactical Embedded Computer Resources, Navy Material Command, Washington, DC, 31 January 1981.

Funded by: Navy

Objectives: "to present the Navy's master plan for the development, acquisition, and life cycle management of tactical embedded computer resources (TECR), including computer hardware and software, used in and in support of Navy combat systems." (p. ES-1)

Methodology: A panel of experts from inside and outside the Navy produced a set of recommendations given in the NECRP final report of October 1978. Those recommendations were incorporated in this study by the Tactical Embedded Computer Program Office, which maintains the Master Plan.

Structure: "The plan addresses problem areas associated with existing and emerging embedded computer systems; outlines plans and strategies for developing and acquiring the embedded computer resources necessary for solving these problems; discusses manpower, personnel, and training requirements; and specifies tasks, milestones, and resources needed to implement the strategies." (p. ES-1)

## E. EVALUATION CONSIDERATIONS

Considering the current status of software metrics, reasonable quantification of benefits is not possible. This is widely recognized; much current research effort addresses the problem. Nevertheless, it is important that some self-consistent methodology be established for judging the value of proposed candidate thrusts. The following approach is suggested.

In selecting candidates, benefits, costs, and interactions among candidates need to be considered. Benefits from a candidate come only after successful R&D and technology transfer, that is, when results are actually used by DoD and its contractors. Costs will be incurred throughout R&D, technology transfer, and operation. Interactions among candidates may make some less desirable (e.g. two efforts addressing the same problem) and some more attractive (e.g. synergy among compatible tools). Each of these topics will be explored in turn: benefits, costs, and interactions.

### E.1 Benefits

Benefits from a thrust activity can occur in the forms of cost savings, time savings, utility enhancement, and risk reduction; when new systems are built and maintained using the R&D thrust products, or when existing maintenance practices are changed to use these products.

Viewed from the initial candidate selection perspective, several steps with varying levels of expected success precede the accrual of benefits. Although the majority of thrusts are highly feasible, research and development may not always produce the envisaged R&D results. Furthermore, transfer of the technology may not succeed for all potential applications within DoD. Thus, after estimating the maximum potential benefits from a thrust activity, the expected benefit could be calculated by multiplying the maximum benefit by the R&D success probability and the technology transfer penetration

percentage. For the most part, estimating such probabilities and percentages with reasonable reliability is extremely difficult or impossible; however, estimates may be more feasible with some thrusts than with others.

R&D success depends on the amount and difficulty of innovation and construction needed. Within a single candidate, several useful results can be sought, each of which has a different probability of success. Likewise, the applicable area of the results may be divided among organizations of differing receptiveness. Figure 8 schematically shows how the present value of expected benefits could be calculated from potential benefits, if quantification were practicable. Even considered qualitatively, however, certain characteristics become apparent. For example, narrowly directed capabilities may be achieved more easily, but their benefits will be less than those of more general capabilities. When evaluating potential thrusts, reviewers should consider the contribution of each projected capability or approach to DoD's software problems.

Much of the benefit will come in the form of cost savings or increased productivity, as new systems are developed less expensively, and as new and existing systems are more efficiently maintained.

When considering the savings benefit of a thrust on new system efforts, a lifecycle model can be used. Figure 9 shows a typical DoD lifecycle cost curve [1]. Since we are concerned with generic effects rather than with a specific project's time and costs, time and costs are measured in terms of the percentage of their totals for the lifecycle. The purpose here is to suggest a framework within which reviewers can evaluate the relative cost savings provided by candidate thrusts. The numbers may not always be defensible, but this section is not an attempt to define exact relative costs of parts of the software lifecycle. Neither is it an attempt to give

<u>Consideration</u>	<u>Measure</u>	<u>Definition/ Formula</u>
R&D	Probability of success in achieving result, R, aimed for by candidate	$Pr(R)$
Technology Transfer	Fraction of potentially applicable areas that use results	$T(R)$
Potential Benefit	Benefit if results used everywhere applicable in DoD	$B(R)$
Expected Benefit	Expected value of benefits from candidate	$E(R)=Pr(R)T(R)B(R)$
Present Value	Sum of the expected values of benefits in each period discounted by interest rate, I, for time t to benefits.	$\sum_t \frac{E(R,t)}{(1+I)}$

Figure 3: Theoretical Calculation of Benefits

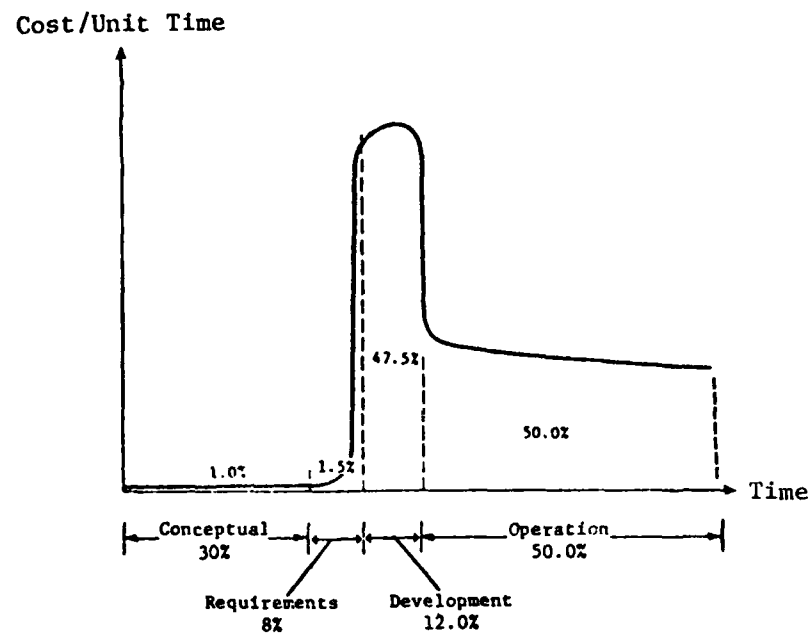


Figure 9: Software Lifecycle [1]

precise definitions for portions of the lifecycle. For a discussion of estimating cost distribution over the lifecycle, see [2].

Some technical thrusts are aimed at specific phases of the software lifecycle: conception, requirements, design, programming, testing, and operation. It may be difficult, however, to associate other thrusts with the lifecycle model. Innovative training concepts, more effective management control of the development process, and the capture of all data relevant to the development and maintenance process are concepts that span the entire lifecycle.

The effect of errors on software cost is estimated in [1] to be half the usual development and operations cost. The cost saving from a thrust that will reduce errors by 20% is calculable at 20% of this potential, or at 10% of lifecycle costs.

Note that elapsed time is also a variable. To a limited extent, the tradeoff desired between cost and time can be accomplished, but will change the shape of the curve. The sooner payoffs occur, the more value they should receive in the evaluation of candidates. The usual present value calculations are relevant here (See Figure 8).

In considering total system cost, all cost elements must be included: technical personnel expenses, management and support personnel costs, direct costs, overhead costs, and (when applicable) contractor fees. In general, total costs vary proportionally with technical personnel costs; however, exceptions should not be ignored when they occur.

Consideration of a thrust's effects on the software lifecycle, either in reducing costs or shortening time, yields a method for gauging its potential cost savings on new systems. New systems, however, are not the only potential beneficiaries of the Software Technology Initiative. Existing systems could also benefit. For example, a technique that transforms code to improve some characteristics

can be useful to existing systems. Benefits to existing systems can be modeled by using the operations phase of the lifecycle.

A thrust's impact on existing systems can be estimated separately; however, it is difficult to quantify the relative value of the same percentage benefits to new systems versus benefits to existing systems. DoD is estimated to spend seventy percent of its software money on maintenance, and this percentage is increasing. The potential effects of a thrust on existing systems enhances its attractiveness.

Although the cost savings framework emphasizes benefits due to reduction in personnel costs, decreased development time, and increased reliability, other potential benefits also exist. These include enhanced functionality, performance, availability, portability, and generality. Portability and generality point to the potential utility of the system in environments or for purposes not originally envisaged. The utility of a system can have four potential sub-utilities depending on its use and its modifiability: (1) used as-is in intended applications, (2) used as-is in other applications, (3) modified in intended application, and (4) modified in other applications.

Cost savings result when a less expensive or quicker way to obtain an existing operational utility is developed. When system function or utility are obtained that would not have been attempted without the prerequisite thrusts, then cost savings do not adequately indicate the full benefits, since one is not doing the same thing less expensively, but rather something new.

Candidates can have results that reduce the risk, that is, the variability or unpredictability in funds, time, or utility. Risk reduction can be very useful to DoD management, quite apart from any reductions in funds or time, or increases in utility.

Lastly, DoD will not be the only beneficiary of the Software Initiative. Other branches of the Federal government, and the software industry as a whole are potential beneficiaries.

## E.2 Cost of R&D Thrusts

The total costs of a candidate for R&D, technology transfer, and operational maintenance are difficult to estimate accurately, although the cost for the first few years of R&D may be relatively easy to estimate. Full development is usually very difficult to estimate. Technology transfer costs require a tentative strategy to be selected and the cost of transferring to the forecasted areas estimated. Operational costs will consist of users' expenses and maintainers' costs for products such as software tools.

Despite the difficulties, part of a candidate's evaluation must be based on its costs. Certainly, exceptionally inexpensive candidates (e.g. superperformer competencies) or ones with unusual expenses (e.g. programmer workstation) need special consideration.

As the STI proceeds, the issue of the costs of thrusts may diminish for efforts of proven promise. For a successful candidate, benefits will far exceed costs. Savings from even a very small reduction in the next decade's software expenses will easily exceed any proposed candidate's costs.

## E.3 Types of Relationships Among Candidates

Thrusts can have several types of relationships that significantly affect decisions concerning their support. A thrust may depend on the prior success of another for its own success, or the benefit from a thrust may be reduced or enhanced by another thrust's success.

Two thrusts can have the same target; the success of one can produce the same benefit as the success of both, or the successes may be additive. Partial redundancy may result from partial overlap of

targets. Combined benefits may come about because one thrust reduces the volume of the problem the other addresses. For example, one candidate might aim at reducing the cost for repairing an error, while another might aim at reducing the number of errors originally committed.

Synergies can also exist among candidates. This is particularly true if the output of one's product is an input of another's. The compatibility of interface between the tools produced by such candidates determines the magnitude of the synergistic effect.

The decision on what set of candidates to select resembles an investment portfolio decision, but has interdependencies more commonly associated with capital investment and project planning decisions. Reviewers should concentrate on evaluating each individual candidate, but consider the interdependencies. To the extent possible, the set of candidates recommended by each reviewer should be internally consistent.

#### E.4 References

- [1] Alberts, D.S., "The Economics of Software Quality Assurance", Conference Proceedings NCC 1976, New York, AFIPS Press, 1976.
- [2] Putnam, L.H. (ed), Tutorial: Software Cost Estimating Lifecycle Control, Los Alamitos, CA, IEEE Press, 1980.

F. SOFTWARE TECHNOLOGY INITIATIVE QUESTIONNAIRE

## SOFTWARE TECHNOLOGY INITIATIVE QUESTIONNAIRE

### Introduction

This Software Technology Initiative Questionnaire is provided for reviewer response. If a separate questionnaire is not available, one can be made by copying Appendix F. The questionnaire will greatly reduce your burden in responding thoroughly, and will speed our analysis of the responses. We suggest that you use the questionnaire to guide your construction of a response, referring, whenever more detail is needed for understanding a candidate, to the appendices: Appendix A for tentatively accepted candidates, and Appendix B for tentatively rejected candidates. Figure 6 summarizes all of the candidates by title, and indicates their tentative classifications. The table of contents can be used to locate the page number of specific candidate descriptions in the appendices.

The questionnaire is divided into 2 parts: (1) a General Questionnaire (which should be filled out by all reviewers) which includes sections on Identifying Information, Problem Areas, Candidates, and Other Comments; and (2) Specific Candidate Questionnaires, one of which should be filled out for each candidate about which a reviewer has knowledge and interest. In addition, reviewers should feel free to add comments on a separate sheet of paper.

Before filling out the questionnaire, reviewers should read Appendix E for an explanation of desired evaluation considerations.

We are aware that many of the questions are difficult and precision may be impossible. Please simply give us the best answer you can.

The results will be analyzed and the analysis used as a basis for discussion on which candidates to include in the Software Technology Initiative.

We are confident that the Software Technology Initiative will benefit from your response. We greatly appreciate your time and interest.

(5/81)

# STI GENERAL QUESTIONNAIRE

### A. Identifying Information

1. Your Name \_\_\_\_\_  
Position \_\_\_\_\_  
Address \_\_\_\_\_  
Telephone \_\_\_\_\_
2. Should these questionnaire answers be considered to be:  
☐ a. The official response of your organization  
☐ b. Your personal response only
3. Your Organization \_\_\_\_\_

## B. Problem Areas

1. Please rate the relative seriousness of each of the 19 major problem areas discussed in Appendix C, as compared to each other. (Order by rating each problem on a scale from one to five, with one being least serious/important and five being most serious/important.)

		SERIOUSNESS				
		Least Serious		Most Serious		
		1	2	3	4	5
<u>Technical</u>						
C.1.1	flawed and conflicting standards					
C.1.2	inappropriate external constraints					
C.1.3	poor definition of goals and measures (e.g., incorrect, unusable success criteria)					
C.1.4	faulty design					
C.1.5	incorrect selection and use of languages and packaged software					
C.1.6	poor use of implementation tools					
C.1.7	inferior testing methodology					
C.1.8	unsatisfactory product evaluation and follow-up					
<u>Managerial</u>						
C.2.1	weak project leadership and coordination					
C.2.2	poor monitoring and prediction of schedules and budgets					
C.2.3	unsatisfactory project control					
C.2.4	flawed methodology for the acquisition process					
<u>Personnel-Related</u>						
C.3.1	problems finding and keeping qualified personnel					
C.3.2	unsuitable competence measures					
C.3.3	poor exploitation of personnel					
<u>Continuity-Related</u>						
C.4.1	ambiguous, unclear, incomplete communication					
C.4.2	slow, outdated communication					
C.4.3	lack of project history					
C.4.4	poor phase-to-phase continuity					

2. Are there any significant problem areas that are not covered by one of these 19 listed areas?        yes        no  
If yes, please list and rank on the five-point scale.

[illegible]

### C. Candidates

#### Instructions for column checklist

1. Please rate this candidate in terms of its overall worth, specifically its contribution toward gaining the desired order-of-magnitude improvement in the software process. Rate the candidate on this five-point scale, with "1" being the lowest worth (little contribution to gaining the improvement) and "5" being the highest worth (high contribution toward gaining the improvement).
2. Do you think this candidate should be:
  - a. accepted (check A)
  - b. rejected (check R)
  - c. accepted with modification (check M)\*
  - d. combined with another candidate (check C)\*\*

\* If c, please note suggested modifications on a Specific Candidate Questionnaire (under #7, "Other Comments"), on a separate sheet, or on a copy of the candidate description found in the appendix.

\*\* Note number(s) of other candidate(s) with which you would combine this candidate.
3. Please check those candidates about which you feel you have sufficient knowledge or interest to fill out the following, more detailed Specific Candidate Questionnaire.

	1					2				3
	OVERALL WORTH					DISPOSITION				
	Lowest Worth		Highest Worth							Will Answer Candidate Questionnaire
	1	2	3	4	5	A	R	M	C	Combine With Other (C)
A.1.1.1 Integrated Software Support Environment										
A.1.1.2 Ada Package Sets for Common Usage Areas										
A.1.1.3 System Dictionary/Directory										
A.1.1.4 Set(s) of Tools Covering Entire Lifecycle										
A.1.1.5 Software Engineer's Support System										
A.1.1.6 Programmer Workstation										
A.1.1.7 Useful Measures of Software Quality										
A.1.1.8 Multiple Representations of Software										
A.1.1.9 Earliest Possible Error Detection										
A.1.1.10 Configuration Independence										
A.1.2.1 Rapid Simulation										
A.1.3.1 Rapid Prototyping										
A.1.3.2 Application Domain Expertise										
A.1.3.3 Data Validation										
A.1.3.4 Built-in Testing										
A.1.3.5 Forgiving Systems										
A.1.3.6 User-Oriented Requirements Interface										
A.1.3.7 Complex Knowledge-Based Systems										
A.1.4.1 Data Flow Approach										

	1					2					3	
	OVERALL WORTH					DISPOSITION					Combine With Other (C)	Will Answer Candidate Question- naire
	Lowest Worth		Highest Worth			A	R	M	C			
	1	2	3	4	5							
TECHNICAL	A.1.4.2 Self-Interfacing Software											
	A.1.4.3 Predicate Approach											
	A.1.4.4 Exception Handling											
	A.1.4.5 Distributed Functions and Resources											
	A.1.4.6 Suitable Communication Intercon- nection											
	A.1.5.1 Transform Software to Improve Quality											
	A.1.5.2 Formal Verification of Large Systems											
	A.1.6.1 High-Confidence Software Testing											
	A.1.7.1 Facilitating System Evolution											
MANAGEMENT	A.1.7.2 Impact Analysis of Proposed Change											
	A.2.1 Acquisition Manager's Support System											
	A.2.2 Software Technology-Compatible Acquisition											
	A.2.3 Technology Transfer in the Software Area											
	A.3.1 Superperformer Competencies											
	A.3.2 Intensive Advanced Programmer Training											
	A.3.3 Programmer Laboratory											
	A.3.4 Personnel Independence											
	A.3.5 Improved Education About Software											
PERSONNEL	A.3.6 User Programming											
	A.4.1 Voice Replace Text											
	A.4.2 Built-In Training and Documenta- tion											

4. Of the candidates tentatively rejected (listed in Appendix B), are there any that you think deserve further consideration at this time?

Yes No

If yes, which?

Why?

---



---



---



---



---



---



---

[illegible]

### SPECIFIC CANDIDATE QUESTIONNAIRE

Candidate Name and Number \_\_\_\_\_

1. Please rate the potential applicability of this candidate's results to, first, your own organization's work and, second, DoD in general:

Own Org.	DoD	
		Very applicable (could solve important/numerous problems)
		Moderately applicable (could solve some problems)
		Limited applicability (might solve few/relatively unimportant problems)
		No applicability (wouldn't solve any problems)

2. How much incremental improvement in the software process do you think could result from this candidate in each time period:

None	Low	Medium*	High*	
				Short-term ( 4 years)
				Medium-term (4 - 7 years)
				Long-term (7 - 10 years)
				Very long-term (more than 10 years)

\* If you checked "medium" or "high" in the short-term, please describe the nature of the benefits you would expect.

---



---



---

3. Please rate the likelihood that the R&D for this candidate will succeed:

- a. very likely \_\_\_\_\_  
 b. moderately likely \_\_\_\_\_  
 c. small chance of success \_\_\_\_\_  
 d. virtually no chance \_\_\_\_\_

4. Please rate the likelihood of successful technology transfer (i.e., the likelihood that applicable areas will actually use it) both within your organization, specifically, and DoD, generally:

Own Org.	DoD	
		Very likely
		Moderately likely
		Little likelihood
		Very unlikely

5. Estimate the potential savings that will result from this candidate's effects on the software life cycle, both by reducing costs and shortening time:

Cost Savings					Time Savings					Phase
Neg*	None	Low	Med.	High	Neg**	None	Low	Med.	High	
										conceptual/feasibility
										requirements
										development
										operations of new systems
										operations of existing systems

\* Negative - will result in cost increase

\*\* Negative - will lengthen time

6. Is your organization supporting any ongoing research or developing any products relevant to this candidate that are not found in the candidate description?  
 Yes \_\_\_\_\_ No \_\_\_\_\_

If yes, please identify (Attach description, if available)

---



---



---

- [illegible]

DATE  
FILMED  
— 8